Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

#115 MAY 1986
$2.95 (3.95 CANADA)

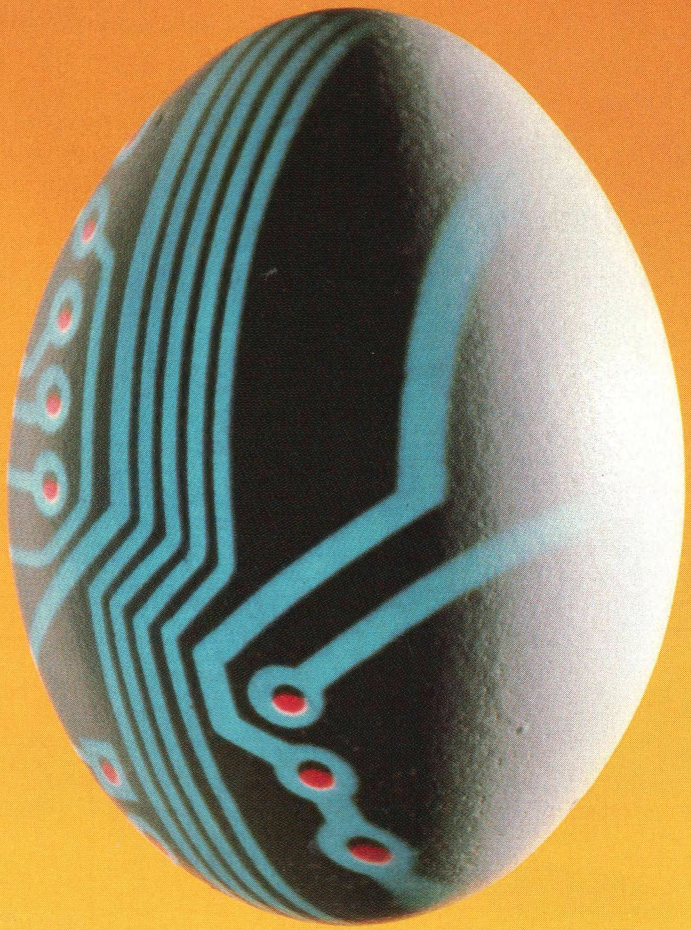
SOFTWARE DESIGN FROM THE OUTSIDE IN

Dan Bricklin's DEMO Program

Cryptographer's Toolbox

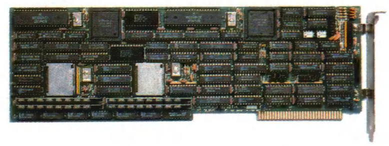
EGA Graphics & Fast PC Graphics

How to Write Memory-Resident Code



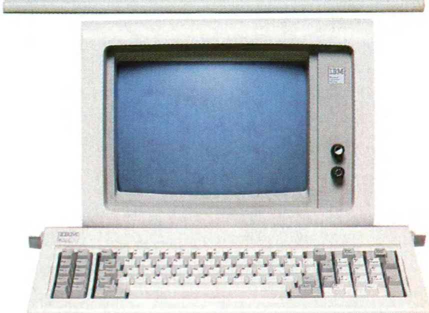
05
0 38351 16562

1+1=3



PC Slave II™

The Ultimate Two-User IBM PC® Card



One IBM PC, XT/AT or compatible, plus one Advanced Digital PC-Slave II, now equals a complete 3-user (or more) system.

Do you need a true multi-user PC system?

If so, read on. The Advanced Digital solution will simply add up in your favor.

To add two or more users to your IBM PC, XT/AT or compatible, just plug in our PC-SLAVE II and two low-cost monochrome monitors and keyboards. You can now run PC-DOS™ or MS-DOS™ on each user with PC-NET™ software to support your network. The end result is a true multi-processor system that allows each user to run independently on their own CPU and memory, yet sharing a common data base.

Do you need networking capabilities?

The realm of IBM networks is only a step away with the PC-Slave II and PC-NET software. An expanding network of multi-user PC's is truly cost-effective since you will be sharing data and communication between users and PCs. File and record locking are also provided so each user can run independently of others while still tied through the network.

Do you need additional users?

Expand your existing supply of PC standalones into multi-user systems by simply adding more PC-Slave II's.

The PC-Slave II features two Intel 80188 CPU's @8Mhz, two 512Kb RAM per CPU and two monochrome/keyboard controllers all on a single board.

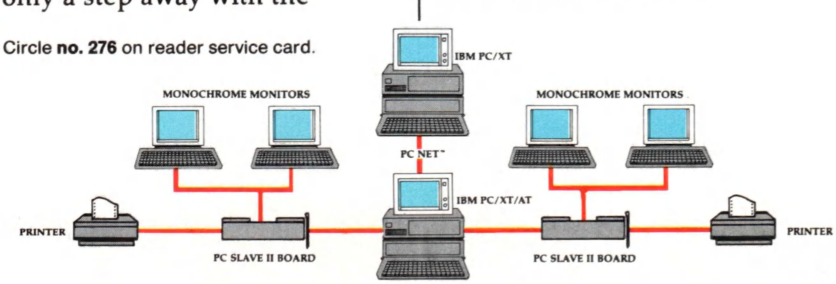
MS-DOS is a registered trademark of Microsoft
PC-DOS, PC-NET are registered trademarks of IBM Corporation
PC-Slave II is a registered trademark of Advanced Digital Corporation

To learn more about the multi-user solution contact:

ADVANCED DIGITAL CORPORATION
5432 Production Drive
Huntington Beach, CA 92649
(714) 891-4004 (800) 251-1801
Telex 183210 ADVANCED HTBH

ADVANCED DIGITAL U.K. LTD.
27 Princes Street, Hanover Square
London W1R8NQ-United Kingdom
(01) 409-0777 (01) 409-3351
TLX 265840 FINEST

Circle **no. 276** on reader service card.



ADVANCED DIGITAL CORPORATION

IBM COMPATIBILITY

. . . at a not so IBM price

TECH TURBO PC/AT
$2299

PRICE INCLUDES:
- 6 TO 8 MHZ 80286 CPU
- 512K
- ONE, 1.2 MB FLOPPY DRIVE
- 8 EXPANSION SLOTS
- 195 WATT POWER SUPPLY
- COMPLETE MS DOS, PC DOS, XENIX COMPATIBILITY
- RUNS LOTUS 123, DBASE III FRAMEWORK AND ALL OTHER POPULAR AT SOFTWARE
- ONE YEAR WARRANTY!!

OPTIONS:

TECH TURBO PC/AT WITH 20MB HARD DISK **$2699**

TECH TURBO PC/AT WITH 20MB HARD DISK, MONOCHROME MONITOR, HERCULES® COMPATIBLE MONOGRAPHICS CARD **$2899**

ALSO AVAILABLE WITH TAPE BACKUPS, MODEMS, LARGER HARD DISKS, AND NETWORKING SYSTEMS.

TECH PC/AT
$1999

PRICE INCLUDES:
- 6MHZ 80286 CPU
- 512K
- ONE, 1.2 MB FLOPPY DRIVE
- 8 EXPANSION SLOTS
- 195 WATT POWER SUPPLY
- COMPLETE MS DOS, PC DOS, XENIX COMPATIBILITY
- RUNS LOTUS 123, DBASE III FRAMEWORK AND ALL OTHER POPULAR AT SOFTWARE.
- ONE YEAR WARRANTY!!

OPTIONS:

TECH PC/AT WITH 20 MB HARD DISK **$2399**

TECH PC/AT WITH 20MB HARD DISK, MONOCHROME MONITOR, HERCULES® COMPATIBLE MONO/GRAPHICS CARD **$2599**

ALSO AVAILABLE WITH TAPE BACKUPS, MODEMS, LARGER HARD DISKS, AND NETWORKING SYSTEMS.

TECH TURBO PC/XT
$1099

PRICE INCLUDES:
- 4 TO 7 MHZ SOFTWARE SWITCHABLE CPU
- 640K
- TWO 360K DS/DD FLOPPY DISK DRIVES
- 8 EXPANSION SLOTS
- 135 WATT POWER SUPPLY
- ONE YEAR WARRANTY!!

OPTIONS:

TECH TURBO PC/XT WITH 20MB HARD DISK **$1599**

TECH TURBO PC/XT WITH 20MB HARD DISK, MONOCHROME MONITOR AND HERCULES COMPATIBLE MONO/GRAPHICS CARD **$1799**

ALSO AVAILABLE WITH TAPE BACKUPS, MODEMS, LARGER HARD DISKS, AND NETWORKING SYSTEMS.

TECH PC/XT
$799

PRICE INCLUDES:
- 4.77 MHZ CPU
- 256K
- TWO, 360K DS/DD FLOPPY DRIVES
- 8 EXPANSION SLOTS
- 135 WATT POWER SUPPLY
- ONE YEAR WARRANTY!!

OPTIONS:

TECH PC/XT WITH 20MB HARD DISK **$1299**

TECH PC/XT WITH 20MB HARD DISK, MONOCRHOME MONITOR, HERCULES COMPATIBLE MONO/GRAPHICS CARD **$1499**

ALSO AVAILABLE WITH TAPE BACKUPS, MODEMS, LARGER HARD DISKS, AND NETWORKING SYSTEMS.

HI-TEK PGS AVT TAXAN IBM AMDEK HERCULES GENOA PARADISE TEAC TOSHIBA HARDWARE/SOFTWARE NETWORKING HAYS AST JRAM HI-TEK PGS AVT TAXAN AMDEK HERCULES GENOA PARADISE TEAC TOSHIBA

PLEASE ALLOW ONE WEEK FOR SHIPPING **VISA, MASTERCHARGE** **TELEX: 272006 Answer Back-TECH** **FAX: 714/556-8325**

TECH PC

TECH PERSONAL COMPUTERS
2131 South Hathaway, Santa Ana, California 92705

714/754-1170

Users only circle No. 279 on reader service card. Dealers only circle No. 245 on reader service card.

CONTENTS

Dr. Dobb's Journal of
Software Tools

Ever wanted to redesign a commercial program's user interface? ▶

Why an advocate of simplicity thinks we need two cursors ▶

Dense graphics

Tools for encryption and decryption ▶

Fast graphics ▶

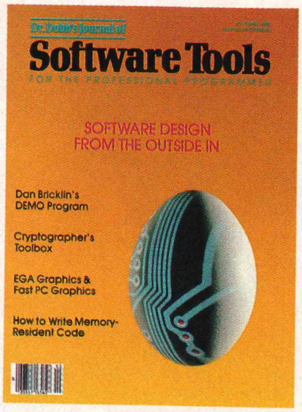
How to write a memory-resident application ▶

ARTICLES

COLUMNS

FORUM

PROGRAMMERS' SERVICES



About the Cover
Photographer Tom Upton redesigned this classic shell.

This Issue
With Bob Frankston, Dan Bricklin won the 1985 ACM Software System Award for inventing a new software metaphor in VisiCalc. Bricklin's latest invention is a tool for allowing software developers to play the what-if game with their own visual metaphors before writing any code. We asked Jim Edlin to use Bricklin's program to demonstrate the process of designing from the outside in.

Next Issue
In June, we'll present techniques for ensuring error-free telecommunications transmission. We'll also take a closer look at Jef Raskin's unusual SwyftCard for the Apple IIe, and introduce two new columns. Structured Programming will concentrate on Algol-derived languages such as Pascal, Modula-2, and Ada; and our editor-in-chief's new column will range from a look at Turbo Prolog to a satire on the issue's theme.


Desk File View Options
Show as Icons
Show as Text
Sort by Name
Sort by Date
Sort by Size
Sort by Type
SC1224
ATARI

THE ST™ COMPUTER LINE FROM ATARI.®

IT'S LIKE GETTING THE POWER AND SPEED OF A FERRARI® FOR THE PRICE OF A FORD.®

When Atari introduced the 520ST™, we set the personal computer industry on its ear.

Nobody had ever produced a machine so powerful and technically advanced for such an incredibly low price. Nobody but Atari has done it yet.

The competition was stunned.

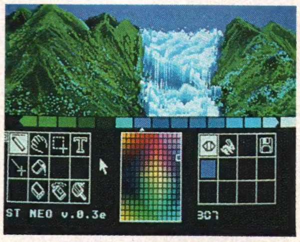
The critics wrote rave reviews.

And consumers were ecstatic.

We could have rested on our laurels, but we didn't.

Instead, Atari extended the ST concept to a new computer called the 1040ST™.

The amazing new 1040ST is even more powerful than the 520ST and years ahead of all the competition at almost any price. The only question in 1986 isn't which company to buy a computer from, but which computer to buy from Atari.



At $799, the 520ST gives you 512 Kbytes of RAM, a high-resolution monochrome monitor, 2-button mouse, and 3.5″ disk drive.

At $999, the 1040ST gives you 1024 Kbytes of RAM, an ultra high-resolution monochrome monitor, 2-button mouse, and a built-in double-sided 3.5″ disk drive, plus built-in power supply. Both the 520ST and the 1040ST can be connected directly to your own color T.V. Or you can add an Atari RGB color monitor to get the sharpest, most colorful images possible. Add $200 for color monitor.

It's simply a matter of choosing which model best fits your needs.

And whether you choose the 520ST or the 1040ST, you'll be getting the power and speed of a Ferrari for the price of a Ford.

In fact, you'll save hundreds and in some cases thousands of dollars over comparable computers. Which is why consumers are still ecstatic. Why the critics are still writing rave reviews. And why the competition is still stunned.

	ATARI® 1040ST*	COMMODORE® AMIGA™	IBM® PCAT™	APPLE® Macintosh™	APPLE IIc®
Price	$999	$1795	$4675	$1995	$1295
CPU	68000	68000	80286	68000	65C02
Speed MHz	8.0	7.16	6.0	7.83	1.0
Standard RAM	1 MB	256K	256K	512K	128K
Standard ROM	192K	192K	64K	64K	16K
Number of Keys	95	89	95	59	63
Mouse	Yes	Yes	No	Yes	Optional
Screen Resolution (Non-Interlaced Mode)					
Color	640x200	640x200***	640x200	None	560x192
Monochrome	640x400	640x200***	720x350**	512x342	560x192
Color Output	Yes	Yes	Optional	None	Yes
Number of Colors	512	4096	16	None	16
Disk Drive	3.5″	3.5″	5.25″	3.5″	5.25″
Built-in Hard Disk (DMA) Port	Yes	No	Yes	No	No
Midi Interface	Yes	No	No	No	No
# of Sound Voices	3	4	1	4	1

Atari 520ST with 512K RAM, $799.

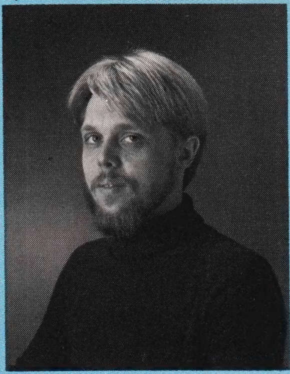
*Connects to standard color T.V. For RGB color monitor add $200.
**With optional monochrome board (non bit-mapped).
***Interlace Mode – 640x400.

ATARI®

POWER WITHOUT THE PRICE.™

Circle **no. 224** on reader service card.

EDITORIAL



I finally hired an editor. Now I can spend my afternoons on the beach as an editor-in-chief ought to. I can also write that back-of-the-magazine column I've been planning—it will appear next month. That's the new guy at the right. I'll let him introduce himself.
—Michael Swaine, editor-in-chief

I'm Nick Turner, the new editor of *DDJ*. I've been a professional consultant for the last few years, and before that I spent three and a half years as a game designer at Atari. (I did Super Breakout, Demons to Diamonds, and Frog Pond, and I worked with another programmer to create Snoopy and the Red Baron—all games for the 2600 VCS machine.) After I left Atari, I joined a start-up firm that promptly went under. I have worked with other companies on a consulting basis.

Now that I'm the editor, I have a chance to help guide the magazine and keep it focused. Always eclectic and pertinent, *DDJ* is more varied and interesting than ever. Professionals experienced in all areas of computer science submit articles that explain their freshest and most innovative ideas. I want to continue the tradition. It's an honor to be involved in a magazine with such a concerned and committed readership.

Now, down to work. We're looking for articles that feature elegant, clean, sophisticated programs. This is your chance to get involved. If you've got a program, utility, or coding technique that is really nifty, will you write it up and share it with the rest of us? Don't forget to send us a disk containing everything that you submit.

If you're in doubt about whether your manuscript would be appropriate for *DDJ*, why not give me a call? You can reach me during the business day (Pacific time) at (415) 366-3600. I'll enjoy discussing our upcoming schedule with you. Here's a quick rundown of what's coming up:

Our September issue, with an author deadline of June 1, will deal with algorithms. I'd particularly like to see something on the recent advances in linear programming. It would also be nice to have some material on real-world algorithms, for example, the algorithms used by robots to navigate complex spaces.

October focuses on the 80286 and 80386 processors. How will they compete with the 680XX and 320XX lines? What makes them more (or less) useful or efficient? How is the upgrade from the 80286 to the 80386 being handled? In particular, I'd like to see some articles with tight machine-language code. October's deadline is July 1.

November will be about graphics. This should be a really fun issue because there's been so much happening recently. I'd like to get some material on the recent advances in surface depiction, ray-tracing, particle systems, or any of the other amazing breakthroughs of the last two years. Manuscript deadline is August 1.

Finally, December will concentrate on operating systems. I'd particularly like material on the very lowest levels of operating-system design—the central kernel—where the nuts-and-bolts task handling takes place. Deadline for December is September 1.

Nick Turner

Dr. Dobb's Journal of
Software Tools

Dr. Dobb's Journal of Software Tools (USPS 3076900 is published monthly by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600, Second class postage paid at Redwood City, CA and at additional entry points.

Article Submissions: Send manuscripts and disk (with article and listings) to the Assistant Editor.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0884-5395**

Customer Service: For subscription problems call: outside CA 800-321-3333; within CA 619-485-9623 or 566-6974. For order problems call 415-366-3600.

Subscription Rates: $29.97 per year, U.S. Foreign rates: $56.97, air; $46.97 surface. Foreign subscriptions must be pre-paid in U.S. dollars, drawn on a U.S. Bank. TELEX: 752-351

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 6686-1520 TELEX: 620430 (WUI)

People's Computer Company

Dr. Dobb's Journal of Software Tools is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

LETTERS

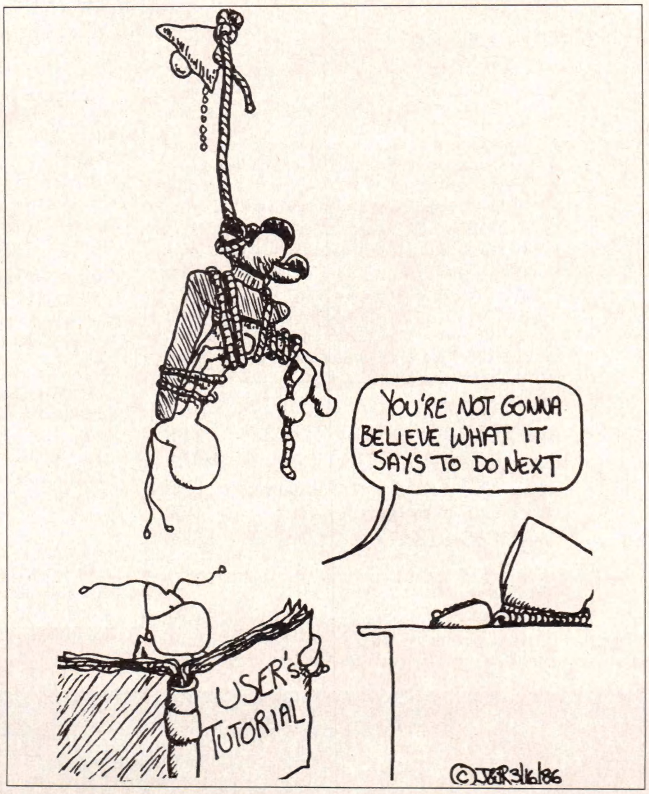
Columns

Dear *DDJ*,
I have a few comments regarding Cortesi's article on disk timings in the September 1985 issue. I ran the BASIC timing program on TESTFILE installed on a RAMdisk. The time was 64 seconds compared to the 122 for a floppy disk. Clearly, BASIC has a considerable overhead in its internal transfer of data, apart from the disk delays. (The copy command copied the RAMdisk file in less than a second.)

Regarding the fast time of the copy command using a floppy, it seems to me that PC DOS applies mainframe techniques to its disk transfers. The fastest way to do a file copy is to transfer cylinders of data at a time. Cylinder transfers do not have to start at the lowest-numbered sector of the cylinder but can start at the first available sector and do a wraparound. The program that reads the disk in this way simply maps the sector onto the appropriate RAM address and counts sectors until all sectors of the cylinder have been read. This is the reason for the speed of a program such as copy.

I cannot let the article's implication that a freshly formatted disk has its first free data area lined up neatly on cylinder 1 pass without comment. The BASIC program and the text of the article imply that to conduct proper timings on a physical device such as a disk, the transfers should line up on complete cylinders. The first free space on a freshly formatted disk is on track 0, sector 4, side 1. Thus the file TESTFILE is misaligned on cylinders and is spread over 17 cylinders, not 16. If, for example, a program were to loop, reading the first nine sectors of TESTFILE in order to give timings of pure reading without head movement, there would in fact be a change of cylinder, resulting in head movement. In the case of the file timings, however, the error is very slight, and the article should be noted carefully by anyone who wishes to write a fast operating system for the IBM PC.

Mike Lawrie
22 Ilchester Rd.
Grahamstown 6140
South Africa



Dear *DDJ*,
I read with interest Dan Daetwyler's comments on converting to DOS 3.0 (16-Bit Software Toolbox, December 1985). As a DOS 2.1 hacker, I have a few comments to make. In spite of his stated distaste for DOS x.0 bugs, I assume that he is using DOS 2.0 because the 20-handle limit per process is present, and enforced, already in DOS 2.1.

The limit is necessary because at address 18h in the Program Segment Prefix, DOS keeps a 20-byte array for translating the user's file handle into another number that DOS uses as an index into its array of files. I doubt that this table is longer in DOS 2.0 because the segment address of the environment is kept at address 2C. Assuming the mechanism to be the same in DOS 2.0 as in 2.1, he is causing a number of potential bugs. First of all, these extra file handles won't be closed automatically when the program terminates or receives a Ctrl-Break. More important, those extra file handles may have their table entries stomped on by whatever DOS does with addresses 2E-5B.

Assuming that DOS 3.1 uses this same mechanism, he might be able to get around the 20-handle limit by maintaining his own longer version of this table elsewhere. In fact, he can do everything with one file handle. For example, to open a file, he could first set [18h]=0FFh. The *OPEN* function call will return file handle 0, and he can move the internal DOS file index from 18h to his own table. Then when he wants to use the file, he can set [18h] to that file index and call DOS with file handle 0, and the DOS function will refer to the desired file. I should note that he will have to close all files himself upon program termination (normal or otherwise). Also, I have not tested this approach, so I give no guarantees. I merely note that COMMAND.COM does a similar trick when bypassing I/O redirection for the "Abort, Retry, Ignore?" message.

Paul Vojta
591 Orange St.
New Haven, CT 06511

Dear *DDJ*,
In your July 1985 issue, 16-Bit Software Toolbox began with the line "One of the most novel features added in Version 2 of MS DOS is the concept of 'installable device drivers.' " I would like to say that this concept may be new and novel for Microsoft and MS

Circle **no. 112** on reader service card.

(continued from page 8)

DOS but it is certainly not a new and novel concept for other operating systems available for microprocessors.

The OS-9 operating system has had the concept of user-installable device drivers since its initial 6809 Level 1 release in 1978. In fact, OS-9 is totally modular in nature and allows for the user addition of new device descriptors, device drivers, and new file managers if required. On top of supporting installable drivers, OS-9 has included the "novel" MS DOS concepts of hierarchical directory structures and pipes. OS-9 also gives full support to I/O redirection, multiprocessing, and multitasking, concepts much more akin to Unix.

OS-9 may not be as well known as MS DOS is, but it does have a large following in the 6809 and 68XXX world today and is growing rapidly. MS DOS has added nothing novel to its OS; it is adding features that are expected and required for an OS in today's world. These features have been around in OS-9 and OS-9/68000 for some time.

Tim Harris
Microware Systems Corp.
1866 N.W. 114th St.
Des Moines, IA 50322

Dear *DDJ*,
This letter has a twofold purpose. The first addresses the source code LJ.C given in the September 1985 16-Bit Software Toolbox, and the second addresses C source code listings for "A New Shell for MS DOS, Part 1: IBM Cursor Control and an Fgets That Edits" in the December 1985 C Chest.

When I tried to run the LJ.C program, it would always hang up on the end of the first file to be printed. I traced the problem to the way that the end of file is found in the *printpage( )* switch. K & R states that the variable *c* should be declared as type *int* so that the end of file (EOF) can be properly stored and tested for. Bringing the source code in line with K & R by changing the variable *c* to type *int* and changing \377 to EOF, which should be defined in stdio.h, cleared up the problem. Listing One, page 68), gives the revised source code with this correction included and modified to allow either Lattice's C compiler or Computer Innovations' C86 compiler to be used.

Also, a minor problem occurred with the source listing of LJ.C. It was hard to distinguish between the letter *l* (ell) and the number *1* (one) in the listing. This was especially troublesome in the escape codes to the HP LaserJet printer, where these two appear often right next to each other. It would be convenient if a typeset that differentiates between these two were used in source code listings.

There is a problem with the source listing for the new shell for MS DOS in December's C Chest that concerns the function *scur( )*. *Pagenum* is passed as an argument but is not declared in *scur( )*. This is a syntax with which I am not familiar, nor is it a syntax that I could find in either K & R or several other books on C. An explanation of this syntax in the next C Chest would be greatly appreciated.

To put the above problems in perspective, I must say that *DDJ* is the best magazine I have found for programmers. I started learning C about three years ago solely by myself from books. It was not until a year and a half ago when I discovered *DDJ* that my real insight into programming started to surface. I attribute a lot of this to your fine magazine and the quality of the articles published. I look forward to each edition of *DDJ* more than I do to any other computer magazine.

Raymond Moon
16005 Pointer Ridge Dr.
Bowie, MD 20716

Ray Duncan replies:
The program as printed in the September 1985 *DDJ* works properly when compiled with Lattice C, as was stated in the column. However, Mr. Moon's changes to the program to make it "standard" are correct, and will be helpful to *DDJ* readers who use other C compilers. An updated version of the program is available in Data Library 2 on the CompuServe DDJ Forum.

Allen Holub replies:
Mr. Moon is referring to:

```
scur( posn, pagenum )
short   posn;
{
}
```

Here *posn* is of type *short int* (the *int* is implied), and *pagenum*, because it's not declared explicitly, is assumed to be of type *int*. That is, *pagenum* is declared implicitly by context. A formal argument list is one of two places in C where the type of a variable doesn't have to be spelled out. (The other is a subroutine call: If the type of the return value of a subroutine isn't declared before the subroutine is used, the compiler assumes that the subroutine returns an *int*.) Implicit declarations are used in the library documentation of most compilers. I looked for a reference in K & R, and all that I could find is: "All variables must be declared before use, although certain declarations can be made implicitly by context" [Kernighan & Ritchie, *The C Programming Language* (Englewood Cliffs, N.J.: Prentice Hall, 1978), 36]. This is not really satisfactory. I'm sure it's spelled out in more detail somewhere in the book, but I couldn't find it. Every compiler that I know of supports implicit argument declarations though.

DDJ

(Listing begins on page 68.)

Those who insist on C compiler performance are very big on Mark Williams.

And the compiler is just part of our total C Programming System.

NEW 3.0 ENHANCEMENTS

These and other powerful utilities now *included* in the C Programming System:

- **make: compiles only what's necessary from multiple modules, a powerful programming discipline**
- **diff: identifies differences between two files**
- **m4: macroprocessor expression editing and substitution**
- **egrep: extended pattern search**
- **MicroEMACS: full screen editor with source**

COMPILER FEATURES

- **Runs under MS-DOS**
- **Full Kernighan & Ritchie C with recent extensions including void and enum**
- **Register variables for fast, compact code**
- **Full UNIX™ compatibility and complete libraries**
- **Large and small memory models**
- **MS-DOS linker compatibility**
- **8087 Support**
- **One-step compiling**
- **English error messages**
- **ROMable code**
- **Linker, assembler, archiver**
- **Extensive third party library support**

csd C SOURCE DEBUGGER

- **Debugs at C source level without assembly language**
- **Separate evaluation, source, program and history windows**
- **Can execute any C expression**
- **Capabilities of a C interpreter, but runs in real time**
- **Set trace points on any statement or variable**

Mark Williams' C compiler has earned a place in some very big companies for some very good reasons: it proves the benchmarks right with the speed, code density, consistent performance and expert support required in professional development environments.

But a total development tool shouldn't stop with compiling. Or go on and on with extras that add up and up.

Only Mark Williams' C Programming Systems *includes* the csd C Source Debugger with true source level debugging to speed your programming job.

And only Mark Williams' new 3.0 version *includes* utilities like "make" to make quick work of even the largest projects.

From source code to final product, only one takes you all the way: Mark Williams' C Programming System. All for only $495. **Ask about our 60-day money back guarantee when you call 1-800-692-1700 to order today.*** You'll be big on the total C Programming system from Mark Williams, too.

*In Illinois call 312-472-6659.

Mark Williams Company
1430 West Wrightwood
Chicago, Illinois 60614

Circle no. **102** on reader service card.

VIEWPOINT

Keep it Quiet

The fact that we can develop technology to make computers speak and understand human speech is not sufficient reason to let them do so. Now that progress has been made in the fields of speech synthesis and recognition, a great rush seems to be on to develop voice applications to capitalize on the technology. Many, if not most, of these applications strike me as worse than useless. There is a simple reason for this: For now and probably for decades to come, voice represents a lousy interface between man and machine.

Consider it from the first direction—computer voice synthesis. Computer speech might be a nice gimmick for a game, but it is totally inappropriate for most serious applications. The purpose of a good user interface is to make things easier for the user. Let's compare a computer voice message with the simple message tone and text on the screen that is common in many of today's applications. When the machine must gain the user's attention, a short tone is more than adequate for the task. Such a tone can register in the user's mind without demanding instant attention, thus allowing the user perhaps to complete a train of thought. Series of tones can indicate degrees of urgency. The important thing is that such a tone leaves the user in control. The user can decide when and how to respond and how much attention to give to the machine.

With a computer voice message, the situation is radically different. The voice not only interrupts whatever the user is doing but also demands that the user interpret the message immediately. Will it become acceptable in our society for a computer to be able to interrupt a conversation between people? I certainly hope not.

A voice message can also be misunderstood easily, especially with many of today's voice synthesizers, which are devoid of expression and difficult to comprehend. Even the good ones may require several repetitions of a message for full understanding—after all, even people often need to repeat themselves. A screen message can be studied at leisure.

Worse yet, there is a tendency to make some machines talk that were clearly meant to remain silent. Most people do not require a lecture on purchasing stamps from the stamp machine in the post office. I do not appreciate a hidden voice telling me to put on my seatbelt, regardless of how nice the voice sounds—a pleasant buzzer and light that can be disconnected easily are sufficient.

The situation with voice recognition is even worse. Let's assume that such recognition systems will be perfected and will achieve very high (say, more than 95 percent) word recognition rates with very large vocabularies and speaker independence. Until that time, voice recognition systems will continue to require significant discipline on the part of the user in terms of care in pronunciation and adaptation to the machine's vocabulary. Although a 95 percent word recognition rate is probably more than adequate when people speak with each other, computers with this ability will still be limited because word recognition is not sufficient. When people communicate with each other, they use many more clues to fill in where word recognition leaves off. Expression, timing, context, eye contact, body language, and even subconscious lip reading are important parts of the communication process.

Until computers can take all of these into account, they will be limited in their capability to understand the user unless the user pays conscious attention to the communication process. This defeats the original goal of making the user interface "friendly." At least with text input, people are accustomed to such discipline. It is probably possible for people to learn to communicate verbally without using nonverbal clues—but is this really desirable? Such discipline would undoubtedly carry over to society as a whole. It is one thing to make machines more human, but should we encourage a process that could make communication between people more machinelike?

Several environmental problems are created when attempting to develop a voice communication system, too. Such systems are often intolerant of background noise, requiring the user to speak loudly into the microphone in order to overcome the noise. An office with such machines would also have significant noise generated just by people talking to their computers.

There are some applications for which even today computer voice systems can be useful. They enable computers to be used by the blind. They can allow a degree of voice control of computers over the telephone. They allow voice mail systems to be built. Future applications should include voice writers (voice-to-text typewriters) and translating systems.

For now, though, the mad rush to incorporate voice systems into common computer applications is premature. Entering information into a database is much more efficient via a keyboard, and system control can be handled very nicely using today's sophisticated user-interface systems and pointing devices.

Someday my argument will no longer be valid. The HAL 9000 computer in the movie *2001—A Space Odyssey* was capable of interpreting subtle nonverbal clues (including lip reading). It was also capable of learning politeness, signaling that it was about to speak with a short, electronic "throat clearing." When such computers are possible, voice communication will become both convenient and necessary—until then, let's keep them quiet.

DDJ

by Daniel Appleman

© 1986 by Daniel Appleman. Appleman is a hardware and software designer at Microlabs, in Sunnyvale, Calif.

Step-by-step tutorial, demo programs with source code included!

Borland introduces Turbo Prolog, the natural language of Artificial Intelligence.

Prolog is probably the most powerful computer programming language ever conceived, which is why we've made it our *second* language—and "turbocharged" it to create Turbo Prolog.™

Our new Turbo Prolog brings supercomputer power to your IBM® PC and introduces you step-by-step to the fascinating new world of **A**rtificial **I**ntelligence. And does all this for an astounding $99.95.

Turbo Prolog is to Prolog what Turbo Pascal® is to Pascal!

Our Turbo Pascal astonished everyone who thought of Pascal as "just another language." We changed all that—and now Turbo Pascal is the de facto worldwide standard, with hundreds of thousands of enthusiasts and users in universities, research centers, schools, and with professional programmers, students, and hobbyists.

You can expect at least the same impact from Turbo Prolog, because while Turbo Prolog is the most revolutionary and natural programming language, it is also a complete development environment—just like Turbo Pascal.

Turbo Prolog radically alters and dramatically improves the brave new world of artificial intelligence—and invites you into that fascinating universe for a humanly intelligent $99.95.

Even if you've never programmed before, our free tutorial will get you started right away

You'll get started right away because we have included a complete step-by-step tutorial as part of the 200-page Turbo Prolog Reference Manual. Our tutorial will take you by the hand and teach you everything you're likely to need to know about Turbo Prolog and artificial intelligence.

For example: once you've completed the tutorial, you'll be able to design your own expert systems utilizing Turbo Prolog's powerful problem-solving capabilities.

Think of Turbo Prolog as a high-speed electronic detective. First you feed it information and teach it rules. Then Turbo Prolog "thinks" the problem through and comes up with all the reasonable answers—almost instantly.

If you think that this is amazing, you just need to remember that Turbo Prolog is a 5th-generation language—and the kind of language that 21st century computers will use routinely. In fact, you can compare Turbo Prolog to Turbo Pascal the way you could compare Turbo Pascal to machine language.

You get the complete Turbo Prolog programming system for only $99.95

You get a complete Turbo Prolog development system including:

- The lightning-fast Turbo Prolog incremental compiler and the interactive Turbo Prolog editor.
- The 200-page reference manual which includes the step-by-step Turbo Prolog tutorial.
- The *free* GeoBase™ natural query language database including commented source code on disk—ready to compile. GeoBase is a complete database designed and developed around U.S. geography. It includes cities, mountains, rivers, and highways, and comes complete with natural query language. Use GeoBase immediately "as is," or modify it to fit your own interests.

So don't delay—don't waste a second—get Turbo Prolog now. $99.95 is an amazingly small price to pay to become an immediate authority—an instant expert on artificial intelligence! The 21st century is only one phone call away.

Turbo Prolog 1.0 Technical Specifications

Programming System Features

- ☑ ***Compiler:*** Incremental compiler generating native in-line code and linkable object modules. The linking format is compatible with the PC-DOS linker. Large memory model support. Compiles over 2500 lines per minute on a standard IBM PC.
- ☑ ***Interactive Editor:*** The system includes a powerful interactive full-screen text editor. If the compiler detects an error, the editor automatically positions the cursor appropriately in the source code. At run-time, Turbo Prolog programs can call the editor, and view the running program's source code.
- ☑ ***Type System:*** A flexible object-oriented type system is supported.
- ☑ ***Windowing Support:*** The system supports both graphic and text windows.
- ☑ ***Input/Output:*** Full I/O facilities, including formatted I/O, streams, and random access files.
- ☑ ***Numeric Ranges:*** Integers: −32767 to 32767; Reals: 1E−307 to 1E+308
- ☑ ***Debugging:*** Complete built-in trace debugging capabilities allowing single stepping of programs.

YES! I want the best

Rush me Turbo Prolog at:

$99.95

To order by phone, or for a dealer nearest you,

Call (800) 255-8008

in CA call (800) 742-1133.

Send me ___ Turbo Prolog at $ ______

Outside USA add $10 per copy
CA and MA res. add applicable sales tax $ ______

Amount enclosed: $ ______

This price includes shipping to all US cities

Payment: VISA MC Bank Draft Check

Credit card expiration date: ___/___

Card # ________________

Carefully describe your computer system: SU11

Mine is: __ 8-bit __ 16-bit

I use: __ PC-DOS __ MS-DOS __ CP/M-80 __ CP/M-86

My computer's name and model is: ________________

The disk size I use is: ☐ *3½"* ☐ *5¼"* ☐ *8"*

NOT COPY PROTECTED
****60-DAY MONEY-BACK GUARANTEE***

Name: ________________

Shipping Address: ________________

City: ________________

State: ________ *Zip:* ________

Telephone: ________________

CODs and purchase orders WILL NOT be accepted by Borland. Outside USA make payment by credit card or International Postal Money Order.

**YES, if within 60 days of purchase this product does not perform in accordance with our claims, please call our customer service department and we will gladly arrange a refund.*

Minimum system requirements:
IBM PC, PCjr, XT, AT, and true compatibles
384K RAM

BORLAND
INTERNATIONAL
Vive la différence

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066
(408) 438-8400 TELEX: 172373

Circle **no. 161** on reader service card.

DDJ ON LINE

The following is a thread of conversation from one of the message boards on the DDJ *SIG*:

Fm: Levi Thomas
76703.4060
I found a press release in the morning mail here at *DDJ* which actually contains some interesting information (rare indeed). Skipping over the fanfare and David vs. Goliath analogy, it says that Microstuf software company has filed suit against Softklone over the "look and feel" of a terminal program that Softklone distributes. The program is called Mirror and is a clone of Microstuf's Crosstalk XVI.

There was trouble of this kind concerning the proposed release of GEM but it was settled out of court, so no legal precedent was set at that time. Has this sort of thing been brought to court before? Softklone's press release goes on to say that Microstuf did not contend that its source code, object code, or user manual were copied, but claimed copyright infringement based on a single input screen. So what do you think? Should the "look and feel" of a computer program be copyrightable?

Fm: Bob Perez 76003,102
Hmm. Well, this area is certainly new, and it's hard to tell what the courts will do with it. Some observations: Close cases tend to be decided on peripheral details in order to bring the balance of evidence in alignment with the decision rendered. It's often said (and the juridicial positivists have not quarreled much with this) that judges make the law, not the legislature, and that they do so according to their perceptions of prevailing moral trends. Hence the liberalization of criminal procedures by the Warren Court during the 1960s and the other end of the spectrum currently underway by the Burger Court. If this is so, then we can expect that judges will soon be bending over backwards to try and fashion some seat-of-the-pants remedies for what is currently being touted as "the threat of piracy." Expect to see it happen first in the copy-protection area (particularly as the issues are raised by the Shrinkwrap licensing notion) and then expect it to expand in other, deeper directions. Perhaps Apple could have presented a good test case with its threats against DRI. . . .

I can immediately see a couple of major strikes against Softklone in this case; points that can only help Microstuf in its effort to sway a judge. Naming their company Softklone was stupid. Period. But then naming their product Mirror was tantamount to mailing a "I Dare You to Sue Me" bumpersticker to Microstuf's corporate counsel. This reminds me of the obscenity litigation that Al Goldstein went though with his magazine. It was bad enough calling it *Screw*, but having it postmarked from Intercourse, Pennsylvania. . . .

Fm: Chris Dunford (INF)
76703,2002
How is the Crosstalk clone conceptually different from, for example, the "store brands" that most of the big supermarket chains sell? You'll find that they sell three or four types of, say, mouthwash: One looks identical to Scope, one to Listerine, and so on. They really go to a lot of trouble to make these things look like the original: same coloring, same bottle shape, same scent, etc. My bet is that, if Softklone has the pockets to put up a decent fight, Microstuf will fold its tent quickly. I've got a hunch it's a scare tactic, and that Les Freed and the gang know they can't win in court. Copyrighting ideas is difficult.

Fm: Bob Perez
I think there are major conceptual differences between the packaging of disposable consumer products and the appearance of a computer program's startup screen. Still, I think the point's worth noting. Maybe a better analogy could be found in the movies. Suppose I release a film that begins with loud blaring trumpets and a screen that looks very similar to the opening screen in Star Wars? Suppose the movie is called WarsClone, and suppose my company's name is LucasLike Films, Ltd. I'd suspect that Lucas's attorneys could start making their retirement plans at that point.

Fm: Chris Dunford
Hmm, not sure that's a good analogy, either <scratching head>. Seems to me that there's a difference between the "feel" of a movie and the "feel" of a program. A film has nothing other than feel; it's an entirely visceral product. If I "clone" the feel of a movie, I have in effect stolen the entire product. On the other hand, a program has more than visceral effects: it does something. And I have just argued myself into a corner, haven't I? The clone-maker has cloned both the feel and the tangible effects of the program. Hmm. I still think that mouthwash provides a good analogy. Not sure I see how cloning mouthwash is conceptually different from cloning Crosstalk.

Fm: Todd M. Roy 70455,140
. . . I think what has happened is that DRI folded too fast to Apple and that a lot of the big name software companies are coming out of the woodwork now to have a try at their imitators.

Fm: Dick Blowers
71555,361
I think of a computer program as a tool because that's the way I use it. I find it hard to think of "using" a novel or a movie.

Fm: Todd M. Roy 70455,140
This sort of stuff doesn't really bother me. It would bother me if one company took another company to court over the basic concept of a program, rather than a clone-like look. For example: Lotus taking everybody who ever made a spreadsheet program to court.

Fm: Robert Grimble
75206,2005
The problem is that you are not copyrighting an idea. You are copyrighting a given display. The problem with the Proctor and Gamble illustration is that bottle shapes are covered by design patents, and there are a lot more on file than there are copyrights on screen displays. Therefore, there is a lot wider coverage by the copyrighted screens, as a practical matter. You are right that the deep pocket theory has a

(continued from page 14)

lot to do with how cases come out, but, when it comes to a user interface, there is no reason why a software developer should not be protected. It costs a lot of bucks and takes more than a little talent for a company to come up with a friendly and useful user interface. That kind of R&D is exactly what patents and copyrights are supposed to protect.

Fm: Bob Perez
Well, these are very tough, interesting issues. On the one hand, I agree that innovation and creativity are stifled when clones are let loose upon the world. On the other hand, Apple's position on their proprietary interface seems to assure that the marketplace will develop in many different, necessarily incompatible directions, at the expense of the very users with whom Apple seeks solidarity. Of course, it's within Apple's rights to milk it while it can. I don't think it's necessarily in their best interests to do so, however.

Fm: Jim Scheef 76505,1351
Your comment on Apple forcing the marketplace to develop many different user interfaces is an important point. Sometime in the future (probably counted in weeks in this industry) someone will realize that people are having a hard time learning to use microcomputers. The reason they will surmise is that the industry has failed to provide a standard interface that people can learn and then use for any computer. This same person, because he or she is also a member of Congress, will then propose legislation to correct this obvious oversight: You have already seen the results of this type of legislation in many other areas. Can you imagine the results if Congress designed icons????!!! Standardization may not be all that far off as many industry watchers believe IBM will move the TopView interface to VM/CMS and then on to all other operating systems as well. Since TopView doesn't use icons, it may not be totally germain to the thread, but Apple and others should realize that no one of them is going to get all of the graphics interface business, so why not agree on common icons that everyone can learn (the way restroom signs have become internationally standard) and use regardless of language.

Fm: Keith Moore
73267,1570
About this icon business: I seem to recall that when Apple came out with the Mac they were talking about all of the symbols stamped into their cabinet, such as the telephone receiver to indicate the modem port. Wherever an international standard for such a symbol existed, they used it. Where there was no standard symbol, they hoped that theirs would be adopted. Why should icons on the screen be different? Also, I seem to remember seeing the trash-can icon on some old Altos software many years ago.

Fm: Jim Scheef
The interpretation of copyright law is an interesting issue. Congress enacted the original copyright law to encourage authors to publish their works; i.e., make the works public. In return, the authors got the rights to publication, performance, etc. for a limited time. With a book it is easy to determine exactly what constitutes the "work"—it's the words and pictures. What is the "words and pictures" of a piece of software? Is it the source code? Remember, to obtain a copyright, the author must "make public" his work. Or is the work the program's output (results)? If it is the results, then does that mean that Lotus owns all of the worksheets I put together using 1-2-3? After all, they are the results or output of the program. Or have I, by writing the spreadsheet, modified the program to produce a different result? I think that's a violation of the license agreement! The issues aren't exactly clear and I doubt we'll resolve the issue here. Going back to the Lotus example, does a program's output stop (for the purposes of copyright) with the intial screen(s) provided by the publisher? If this is true, then how will anyone ever protect an artificial intelligence program that modifies it's output depending on the responses of the user? How would you know if someone had cloned such an AI program? Even the author might not know all of the program's possible outputs!

Fm: Robert Grimble
I recall that one judge commented in a computer case that "if it looked the same it was the same." There is no reason why you can't copyright an input screen, and game manufacturers shoot film of various game screens and copyright them. I'm not a copyright expert, but I think that to win a copyright case you have to show actual plagarism, which is unlike patents, where all you have to show is the use of the same device, process etc.

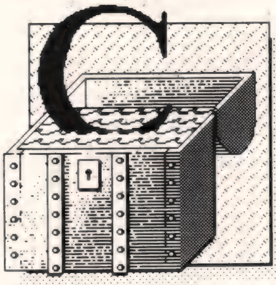
Fm: Ran Talbott
(ProgGnosis) 70506,60
As I (mis?)understand the law, there are two separate issues here: The source code is clearly the "words and pictures" part. You don't send diskettes or PROMs to the copyright office: They are considered electronic representations of the original source. The appearance of the screen is treated as a separate work: Witness the Pac-Person clones that got stomped even though they ran on different machines and couldn't possibly infringe the source copyright. If Apple had successfully sued over their screen copyright, it could conceivably become illegal for anyone to publish, say, a book or magazine article containing a screen representation of a spreadsheet. A win by Apple would say: "Yes, your names (data) are different, but overall it looks like what I sent to the copyright office. Pay up."

Fm: Todd M. Roy
Another interesting point would be to make any copying of a program illegal. Since a program must be loaded in from a floppy disk to computer memory, this is a form of copying, and hence if all copying were illegal, running the program would be illegal. Don't laugh, this isn't even a new point.

DDJ

Accessing IBM Video Display Memory and a Microsoft Bug

by Allen Holub



Strictly speaking, it's a real no-no to read and write directly to or from the IBM PC's display memory. You're supposed to go through the ROM BIOS routines. Unfortunately, though the BIOS is faster than DOS, sometimes it's still not fast enough. This month I'm going to look at a set of C subroutines that talk directly to the IBM PC display memory. These routines are blindingly fast—blink, and you've missed the action. They'll work on all versions of the PC that exist now (at least all versions that use a monochrome adapter or equivalent—the Hercules graphics card is an equivalent), but they may not work on future hardware. In the process of writing these routines, I came across a bug in the way that the Microsoft compiler handles *far* pointers. I'll talk about this as I explain the code.

The routines supported are all in video.c (Listing One, page 72). They are:

```
void    setcur ( row, col )
int     row, col;
```

which positions the cursor at the indicated row and column;

```
void    getcur( rowp, colp )
int     *rowp, *colp;
```

which gets the current cursor position (notice that *rowp* and *colp* are pointers to places where the cursor positions will be put;

```
void    d_putc( c, attrib )
int     c, attrib;
```

which writes a single character with the indicated attribute (more on attributes in a moment) at the current cursor position and then advances the cursor. Wraparound to the next line is supported; however, if you write past the bottom-right corner of the screen, the cursor will roll up to the top-left corner (that is, the screen doesn't scroll). Three control characters are supported: *\r* gets you to the beginning of the current line, *\n* gets you to the same column on the next line, and *\b* goes backward one character. All other control characters print as funny-looking IBM graphics characters of some sort (smiley faces, hearts, or whatever).

The next routine

```
void    d_puts( str, attrib )
char    *str;
int     attrib;
```

writes out a string, with the characters having the indicated attribute. You must use *\r\n* to get to the left edge of the next line.

Finally,

```
void    clrs( attrib )
char    *str;
```

clears the screen by filling it with blanks having the indicated attribute (that is, a reverse-video attribute will fill the screen with a white background and nothing in the foreground; an underline attribute will fill the screen with underscores—that is, an underlined ' ').

Part of the header from Listing One is reproduced in Table 1, page 20. *NUMROWS* and *NUMCOLS* define the screen size. *VIDBASE* is the base address of the display memory used by the monochrome adapter. The address is in cannonical form—that is, *0xb000000* is actually address B000:0000.

Three basic attributes are supported: *NORMAL* video, *UNDERLINED* characters and *REVERSE* video. These are mutually exclusive (you can't have an underlined, reverse-video character). Two modifiers are supported, however: *BLINKING* and *BOLD*. The former causes the character to blink, and the latter causes it to be printed at high intensity. The modifiers may be ORed with each other and with any of the basic attributes. For example, a blinking, high-intensity underlined character has the attribute *BLINKING|BOLD|UNDERLINED*.

Characters are stored in memory as 16-bit objects. The low byte holds the ASCII code, and the high byte holds the attribute. The *CHARACTER* type is a structure that lets you access the ASCII code and attribute independently, without doing a shift and mask operation. If *p* is a pointer to a *CHARACTER*, then *p->letter* changes the ASCII field and *p->attribute* changes the attribute field.

DISPLAY (on line 26) is a 25 × 80 array of *CHARACTER*s; it's the entire display area. *Screen* is a pointer to a *DISPLAY*. The *far* keyword is supported by the Microsoft compiler to ease mixed-model programming. *Screen* is declared here as a *far* pointer to an array—that is, the array can be anywhere in memory and *Screen* is a 32-bit wide pointer to that array. If you're using the Lattice compiler, dispense with the *far* keyword and compile the module using the large data model.

Note that *Screen* is a pointer to the entire array, not to one element of it. Consequently individual elements of the array must be accessed with square brackets. For example, a boldface character can be written at row 5, column 10 with:

```
(*Screen)[5][10].letter = 'c';
(*Screen)[5][10].attribute =
                    NORMAL|BOLD;
```

C Programmers! First database written exclusively for C is also royalty free

"If you are looking for a sophisticated C Programmers database, **db_VISTA** is it..."
Dave Schmitt, President of Lattice, Inc.

Designed exclusively for C, db_VISTA™ is a royalty-free programmers' DBMS. Both single and multi-user versions let you take full advantage of C through ease of use, portability and efficiency.

Written in C for C Programmers

All functions use C conventions so you will find db_VISTA easy to learn. db_VISTA operates on most popular computers, and because it is written in C it can easily be ported to most computers.

Royalty-Free, You only pay once

Whether you're developing applications for a few customers, or for thousands, the price of db_VISTA is the same. If you are currently paying royalties for a competitor's database, consider switching to db_VISTA and say goodbye to royalties. To help you make the change over to db_VISTA, file transfer utilites are available for dBASE, R:base and ASCII files.

More from your database applications with source code

Source code includes all db_VISTA libraries and utilities.

1. Recompile our run-time libraries utilizing non-standard compiler options.
2. Create a debugging library including a function traceback by activating preprocessor commands embedded in the source code.

Multi-user and LAN capability

Information often needs to be shared. db_VISTA has multi-user capability and supports simultaneous users in either multi-tasking or local area networking environments, allowing the same C applications to run under UNIX and MS-DOS.

Faster execution without data redundancy

Less data redundancy means reducing disk storage requirements and maximizing data access performance. A customer evaluating a leading competitor's product prior to purchasing db_VISTA benchmarked db_VISTA's retrieval time to be 276% faster than a leading competitor.

Complete documentation included

User manual contains 193 pages, 8 diagrams, 10 tables, appendices, an extensive index, plus a database application example. 9 chapters with complete instructions.

Introducing db_QUERY™

With db_QUERY you can ask more of your database. db_QUERY is a linkable, SQL-like ad hoc query and report writing facility. It's also royalty-free.

30 day Money-Back Guarantee

We wish to give you the opportunity to try db_VISTA for 30 days in your development environment and if not satisfied return it for a full refund.

Special Offer

Free Falcon hard disk controller with purchase of db_VISTA Multi-user with source.*

Price Schedule

	db_VISTA	db_QUERY
Single-user	$195	$195
Single-user with Source	$495	$495
Multi-user	$495	$495
Multi-user with Source	$990	$990

Free 90 days application development support
All software not copy protected.

Call Toll Free Today!

To order or for information, call TOLL FREE 1-800-843-3313, at the tone touch 700-992.
VISA and MASTERCARD Accepted

Read what others say about db_VISTA . .

"If you are looking for a sophisticated C programmers database, db_VISTA is it. In either a single or multi-user environment, db_VISTA lets you easily build complex databases with many interconnected record types. The multi-user implementation handles data efficiently with a LAN and Raima's customer support and documentation is excellent. Source code availability and a royalty-free run-time is a big plus."
Dave Schmitt, President
Lattice, Inc.

"Not 'yet another user-friendly database', it is a DBMS aimed at the technical C programmer instead of the non-technical end-user".
Hal Schoolcraft, Data Based Advisor
March, 1985

"On the whole, I have found db_VISTA easy to use, very fast with a key find, and powerful enough for any DBMS use I can imagine on a microcomputer".
Michael Wilson, Computer Language
September, 1985

db_VISTA Version 2.1 Database Management System for C

Database Record and File Sizes

- Maximum record length limited only by accessible RAM
- Maximum records per file is 16,777,215
- No limit on number of records or set types
- Maximum file size limited only by available disk storage
- Maximum of 255 index and data files

Keys and Sets

- Key lengths may be a maximum of 246 bytes
- No limit on maximum number of key fields per record - any or all fields may be keys with the option of making each key unique or duplicate
- No limit on maximum number of fields per record, sets per database, or sort fields per set
- No limit on maximum number of member record types per set

Utilities

- Database definition language processor
- Interactive database access utility
- Database consistency check utility
- Database initialization utility
- Multi-user file locks clear utility
- Key file build utility
- Data field alignment check utility
- Database dictionary print utility
- Key file dump utility
- ASCII file import and export utility

Features

- Multi-user support allows flexibility to run on local area networks
- File structure is based on the B-tree indexing method and the network database model
- Run-time size is variable - will run in as little as 64K, recommended RAM size is 256K
- Transaction processing assures multi-user database consistency
- File locking support provides read and write locks on shared databases
- SQL based db_QUERY is linkable and royalty free
- Operating system support for MS-DOS, PC-DOS, Unix, Xenix or Macintosh
- C compiler support for Lattice, Microsoft, DeSmet, Aztec, Computer Innovations, Xenix and Unix
- File transfer utilities for ASCII, dBASE and R:base optional

Independent Benchmark Results

Eleven key retrieval tests on sequentially and randomly created key files. Benchmark procedure adapted from "Benchmarking Database Systems: A Systematic Approach" by Bitton, DeWitt, and Turbyfill, December, 1983

Total Retrieval Time of 11 Tests	
db_VISTA	:671.24
Leading competitor	:1,856.43

RAIMA™ CORPORATION
12201 S.E. Tenth Street
Bellevue, WA 98005 USA
(206) 747-5570
Telex: 9103330300 BCN RIVERTON

1 (800) 843-3313
at the tone touch 700-992

MasterCard VISA

* Limited offer available to end-user purchases directly from Raima Corporation.

Circle **no. 206** on reader service card.

C CHEST
(continued from page 18)

The parentheses are required because the dot (.) has higher precedence than the *. Without the parentheses, the expression evaluates to:

```
*( Screen[5][10].letter ) = 'c';
```

which is meaningless given the declared type of *Screen* (the compiler should kick out an error message).

Because *Screen* is never modified, it should be possible to replace it with a constant. The procedure can be explained more easily by looking at 8080 code rather than 8086 code. If you want to modify memory location *0x100* in an 8080, you can say:

```
#define MEM ((char *) 0x100)
x = *MEM;
```

Here, you're saying treat the number *0x100* as if it were the contents of a character pointer, so **MEM* is the thing pointed to by the implied character pointer. The problem is more complicated with an 8086 because of memory segmentation, but the syntax is more or less the same. Instead of saying:

```
DISPLAY far *Screen =
                    (DISPLAY far *) VIDBASE;
(*Screen)[5][10].letter = 'c';
```

you should be able to say:

```
#define SCREEN ( (DISPLAY far *)          /
                                 VIDBASE )
(*SCREEN)[5][10].letter = 'c';
```

SCREEN is treated as if it were the contents of a variable of type *(DISPLAY far *)*. You can also move the * that precedes the use of *SCREEN* into the macro:

```
#define SCREEN ( *( (DISPLAY far *)
                                 VIDBASE) )
SCREEN[5][10].letter = 'c';
```

As it turns out, the foregoing works fine on the Microsoft compiler provided that the indexes are constants. That is:

```
SCREEN[5][10].letter = '*';
```

causes the following code to be generated:

```
mov   bx,-20480
mov   es,bx
mov   bx,820
mov   BYTE PTR es:[bx],42
```

where *−20480* is *0xb000*, *820* is the offset to row 5, column 10: *2*((5*80)+10)* and *42* is the *. Unfortunately, strange things happen when you try to replace the constants with variables:

```
int     Row = 5;
int     Col = 10;

SCREEN[ Row ][ Col ].letter = '*';
```

generates the following:

```
mov    _Row,5
mov    _Col,10

mov    ax,160
imul   _Row
mov    si,ax
mov    bx,_Col
shl    bx,1
mov    BYTE PTR
              [bx-1342177280][si],42
```

I don't know where that *−1342177280* came from, but it's wrong. When you use the *Screen* variable rather than the *SCREEN* constant, everything works fine, though.

The final thing worth mentioning here is the *clrs( )* routine (on line 114 of Listing One). Here, rather than use *Screen*, I've speeded up the code by declaring a pointer to an individual *CHARACTER* and then cleared the *Screen* array as if it were linear, rather than two-dimensional. This saves all the multiplies implicit in an operation involving square brackets.

Note that redirection obviously won't work when you use these routines for output, but redirection doesn't work when you use the BIOS routines either. Also, be careful about mixing these routines with normal BIOS calls. If the cursor is updated by the BIOS, then the variables *Row* and *Col* used in video.c won't correspond to the real cursor position any more (the BIOS won't update them for you) and strange things will start happening on your screen. In spite of these problems and the lack of portability, direct screen writes are very nice to have, especially if you're creating a full-screen editor or similar program, in which you have to update the screen fast.

```
#define NUMROWS       25
#define NUMCOLS       80
#define VIDBASE       0xb0000000   /* Base address of video   screen
                                    * in canonical form.
                                    */

#define NORMAL        0x07       /* Basic attributes. Only one      */
#define UNDERLINED    0x01       /* of these may be present.        */
#define REVERSE       0x70

#define BLINKING      0x80       /* May be ORed with the above      */
#define BOLD          0x08       /* and with each other             */

/*------------------------------------------ */

typedef struct

typedef struct
{
        char    letter;
        char    attribute;
}
CHARACTER;

typedef CHARACTER        DISPLAY[ NUMROWS ][ NUMCOLS ];

static DISPLAY far *Screen = (DISPLAY far *) VIDBASE;
```

Table 1

Beating Dead Horses

At the risk of being tedious, I'm bring-

ing up the subject of lvalues and pointers once more (this is the last of it, I promise). Steve Hersee (VP of marketing at Lattice and chair of the ISO working group in C) sent me the following paper. It was written by Gary Merrill and Francis Lynch, also of Lattice. Steve points out that the letter was submitted to, and adopted by, the ANSI comittee, so I guess the other side wins. I hate to give up $*(\ (T)P\ )++$, though. If you haven't already, you should read last month's column for background. The letter refers to sections in an early draft of the standard that I don't have a copy of. It's pretty understandable, though, even without the standard in front of you, and it makes some good points about lvalues.

Consequences of the Proposed Standard

The decision to allow an expression of the form $(T)P$ to be an lvalue "for purposes of additive operations on P" (when P is an lvalue of pointer type and T is the name of a pointer type) appears to preclude the development of any coherent semantics for the C language and to prevent its implementation on a significant class of machine architectures. It has the virtue, however, of bringing to light a number of places in the proposed standard where a lack of rigor and precision tend to leave the reader in a state of confusion. While this decision may initially appear plausible and even intuitively correct, the price one must pay for it quickly begins to seem excessively high, if not prohibitive.

Consider the following: Assume that P is an lvalue of pointer type and T is the name of a pointer type. Then we know by the above principle (see Section 4.2.4) that, within the context $++(T)P$ (and also within the context $(T)P++$), the expression $(T)P$ is an lvalue.

On the other hand, we know from Section 4.2.1 that $++(T)P$ is equivalent to $((T)P += 1)$. So, if $++(T)P$ is syntactically (and semantically) acceptable, then so is $((T)P\ +=\ 1)$. We also know (Section 4.14) that $+=$ requires an lvalue as its left operand, so everything appears to be OK so far, so long as we agree that in the context $((T)P += 1)$ the subexpression $(T)P$ is being used "for purposes of additive operations on P."

Now we also know (Section 4.14.2) that $((T)P += 1)$ differs from $((T)P = (T)P + 1)$ "only in that" $(T)P$ "is evaluated only once," and if this is the only difference, then since the former is acceptable syntax, so is the latter. That is, we have just been forced to regard

(1) $(T)P = (T)P + 1;$

as a perfectly acceptable C expression. This requires us, of course, to agree that the first occurrence of $(T)P$ in this expression is an lvalue.

Now come the interesting questions: if (1) is correct, then on what grounds can we possible forbid

(2) $(T)P = 1 + 1;$

or, in general,

(continued from page 21)

```
(3) (T)P = x;
```

Our only hope is to argue that in (2) and (3) there is no "purpose" for an additive operation on *P*. So we have to be willing to say that (1) involves an additive operation on *P* while (2) does not.

But this does not really make things any clearer. How, for example, do we tell in general whether an expression is being used "for the purposes of additive operations on *P*?" The context of (1) is particularly simple. Shall we impose the arbitrary condition that, in order for *(T)P* to be considered an lvalue in the expression

```
(4) (T)P = E
```

that it occur as a subexpression in *E*? What is the possible justification for this ad hoc exception to an otherwise elegant and uniform grammar? Certainly it is counterintuitive and terribly confusing to a user of the language to insist that the left occurrence of *(T)P* in (1) is an lvalue but its left occurrences in (2) and (3) are not. Is what we gain worth the imposition of such an arbitrary exception to our grammar and the resulting confusion?

Note, incidentally, that the simple approach just suggested will not work, for the mere occurrence of *(T)P* within *E* does not ensure that it is used "for the purposes of additive operations on *P*." It may occur in *E* in such a way that its value is never used or is never used in assigning the new value of *(T)P*.

There is another reason that the simple approach of requiring *(T)P* to occur within *E* will not work. Suppose that *P* is a global variable and we have defined a function *f* such that:

```
(T) f( )
{
      return( (T)P + 1 );
}
```

Now consider the expression

```
(5) (T)P = f( );
```

Certainly we are using (5) "for the purposes of" an additive operation on *P*. Can a compiler be expected to detect this purpose? Even if the definition of *f( )* and (5) occur in different source files? To answer yes to these questions is to demand serious revision in the way we conceive of C.

The Semantic Problem

Now exactly what is the problem in all of these cases? It is that an attempt has been made to make a single exception to an otherwise general and formally correct rule: that a cast expression (such as *(T)P*) is never an lvalue. There are very good and well-known reasons for this rule, and they follow directly from the semantics for C.

The motivation for this attempt appears only to be a desire to ensure that certain dubious (at best) coding practices [Humph!—AH] will be tolerated by any compiler conforming to the proposed standard. It may be argued, for example, that if the exception is not made, then a certain amount of existing code will fail to compile under a standardized compiler. This may be true, but in light of the consequences of embracing the exception, it can hardly be looked upon as a compelling argument.

The totally misguided nature of this attempt at construing *(T)P* as an lvalue in a context-dependent way can be seen in the examples described above. The most immediate problem is the appeal to "purposes" in a description of the exception. Expressions of a programming language do not possess intrinsic "purposes" (though those who write them may have a purpose in doing so, of course). An appeal to "purposes" has no place in the definition of a programming language or in the document purportedly describing its standard. As it now stands, the proposed attitude toward considering *(T)P* an lvalue is very close to saying that "*(T)P* will be considered an lvalue when the programmer wishes it to be (ie., when it is his or her purpose to use it as an lvalue), and otherwise it will not be an lvalue." Given the use of "purposes," there can be no pretense at creating a formally precise standard for the language that will enable compiler writers to implement the standard in an objective manner.

The Machine Problem

Another difficulty with considering *(T)P* an lvalue (for any purpose) is that some machine architectures require different representations for pointers to different classes of objects. Clearly, if *(T)P* forces a conversion of *P* to a different representation, there is no justification whatever for regarding the result as an lvalue. On word-addressed machines, the example shown in 4.2.4 cannot be implemented because character pointers are simply not congruent with pointers to long integers. [A better example is the 8086 where, in the medium model, a pointer to data can be 32 bits when a pointer to a subroutine is 16 bits.—AH] Is it the intention of the proposed standard to prevent the implementation of its definition of C on such machines? The usefulness of C is such that we should make every attempt to allow its implementation on as many machines as possible; indeed, one of the principal purposes of a high-level language is to insulate programmers from the idiosyncrasies of the underlying hardware. While no attempt to embrace the entire spectrum of existing architectures is likely to be successful, surely C should not be limited to those machines that use a single pointer representation for all objects? It is true that word-addressed machines present serious problems for the large body of existing code that as-

sumes a single pointer type. The standard, however, should be sufficiently general that it is independent of any assumptions of this kind.

Possible Solutions

What is needed?

1. Replace the clause "for purposes of additive operations on *P*" with a formally precise description of cases in which *(T)P* is an lvalue. We do not believe this can be done without doing violence to a large portion of the sematics of C. The price is simply too high.
2. Treat *(T)L* (where *L* is an lvalue) as always being an lvalue. The arguments are well known and appear to be compelling.
3. Give up the equivalences between *(++E)* and *(E+=1)* and between *(E+=1)* and *(E=E+1)* that allowed us to generate the problematic examples above. But we cannot simply "give up" these equivalences since they are a consequence of the semantics for ++, +=, +, and =. To give them up would not only be counterintuitive but also would require a gross revision of the semantics of the operators of C. This route is impossible.
4. Conform to the currently accepted principle that *(T)L* is never an lvalue. This now appears to be the only reasonable course, and our strong recommendation. The price we pay for it is that some previously existing code that has been accepted by some compilers will no longer be accepted by a standardized compiler. But it is hopeless and pointless for a standard to seek to preserve the acceptability of all previously existing code. The price we must pay for embracing this alternative is insignificant in comparison to what we would have to pay for any of the others.

June 8, 1984
Gary H. Merrill and
Francis L. Lynch

Putting the Shell Level into the Unix Prompt.

The MS DOS shell presented in the January-March C Chests makes the current shell level available in an environment variable. Unix doesn't. Nonetheless, in Unix it's possible to create an almost infinite number of nested shells, and it's occasionally useful to know the current level of shell nesting. The current shell level can be printed by the Unix C Shell. This is done using the program in Listing Two, page 74, in conjunction with the shell script in Listing Three, page 74. Listing Three shows additions you should make to your .cshrc file.

DDJ

(Listings begin on page 72.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Human Interface Design: *From the Outside In*

by Jim Edlin

DEMO gives you a chance to noodle through a dozen different approaches before the programming meter starts ticking.

I got into the user-interface design business by accident. Early in 1980, while searching for a word processing program for my own use, I came across one that stood well out from the herd. Naturally, I arranged to get a copy. I wanted it for drafting a sizable manuscript I had just contracted to do. What I didn't understand in making my impulsive commitment to this program was that it was still under development. I became an unwitting beta tester.

I also soon discovered that this thing that looked so beautiful and seductive on first meeting revealed some disagreeable imperfections at a closer view. The developers offered to correct some of the design flaws. Soon I became their impromptu adviser on design improvements—for example, in the case of a tab-stop setting sequence that came up with a "Column Number? ___" prompt, I told them: "When someone wants to set a tab stop, he may have no idea of what column number he wants it in. What he knows is that he wants the tab stop here," and I pointed. The tab-stop interface was changed to one where the user pointed via cursor to the desired column for the tab stop.

Since then, my involvement in user-interface design has become more formal. For one thing, I started a software company—Bruce & James Inc. My big design effort so far has been B&J's flagship product, the Wordvision word processor (actually a descendant of that program I volunteered suggestions for earlier). The design work for Wordvision extended throughout the program's 15-month development cycle and has recently resumed in preparation for a new edition.

I have also wrestled with interface designs for several other Bruce & James products, released and unreleased, and have evaluated interfaces for many other companies' products in the course of considering them for publication, writing about them, or informally advising their developers.

Most recently I have organized a company with the explicit mission of assisting others with user-interface design. During all my work with user interfaces, I have been seeking tools and techniques to help with the job. I've wanted to find things that would enhance my creativity and productivity, just as a word processor has aided my writing. I have tried out numerous possibilities. Until recently, I have found them all wanting.

Pretenders

My first tool was the BASIC language. I wrote BASIC programs to make prototype screen designs and to change the display in response to keystrokes. This was laborious, particularly when the screen used display features beyond simple printing of text and when it needed a lot of color and attribute painting. BASIC (particularly earlier versions) just isn't built for such jobs.

The next approach was inspired when I showed one of my BASIC prototypes to Peter Jennings, a founder of the former software giant VisiCorp. He suggested I use "storyboards." The idea seemed an obvious fit because I had worked in TV advertising, where the storyboard—a series of drawings sketching out the sequence of key im-

*The Softer Interface Inc., 2355 Leavenworth, Ste. 103
San Francisco, CA 94133*

ages in a commercial—is one of the fundamental tools of the trade.

Eventually, I used the storyboard approach to develop many of the preliminary designs for Wordvision. At first, however, I found storyboards unsatisfactory for the same reason that, after adapting to word processing, I now find writing on paper unpleasant. Furthermore, advertising storyboards portray generalities of visual concepts; in software I wanted to work down to the level of fine detail. Ink on paper was not malleable enough. It also didn't have a useful dimension for portraying and experimenting with flow of control—that part of the interface dealing with how the parts relate and how users move among them.

The flow-of-control question forced me to my next approach—a repeatedly reinvented series of forms on paper. These contained spaces for writing in a description of the screen display; an explanation of what events could cause the display of that screen; a list of user options while on that screen; a description of other screens or actions to which these choices would lead; and a space to describe audio, animation, and other happenings, such as use of peripheral devices.

The form kept getting revised because I continued to run up against situations that demanded entry of information for which there were no spaces on the current version of the form. Nevertheless, the forms acted as a sort of linked list and furnished fairly definitive instructions for programming. But they failed to give an overall sense of how the program would actually operate and "feel."

Eventually the forms gave way to a system of diagrams and notation formalized by a coworker. We maintained these in pencil to allow for the inevitable frequent revisions. The resulting binder became the essence of a user-interface specification. It did the recording job; but as a designer's tool, it was as cumbersome as the intricate notation used to record dance compositions. It's difficult for me to imagine a choreographer finding it more productive and rewarding to scribble pages of intricate notations than to observe action on a stage as he instructs dancers to try different moves and sequences. Likewise with interface design.

In 1984, I finally found a software tool for designing interfaces—Window Panes by Jim Canright ($160 from Softright, P.O. Box 132, Beaverton, OR 97075; (503) 641-4072). Window Panes added a lot of leverage to my design toolbox. It consists of two subprograms: a display editor to create screen designs as either full or partial windows, and a "walker" to create and edit sequences for the windows created with the editor. The sequences can be based on keystrokes, on elapsed timings, and so forth. With Window Panes, it is possible to develop prototypes of both screen designs and flow of control in reasonable detail.

Window Panes had its drawbacks, though. First, because it was difficult to use, I never did quite master its intricacies. Second, every screen or window overlay display called for a separate disk access. Although it operated speedily with a RAMdisk, the business of getting it set up with access to the right files every time was an annoyance. I don't want to downgrade the value of Window Panes; I considered it a true boon when it arrived, but in actual use I found it frustrating.

The version of Window Panes I used is two years old. There is a new version out that probably has improvements, and Canright tells me a still-newer version should be out by the time this article is published.

After Window Panes, the next tool I adopted was Saywhat ($39.95 from The Research Group, 88 S. Linden Ave., South San Francisco, CA 94080; (415) 571-5019). Saywhat is a screen generator originally designed for building displays for dBASE II applications. Later The Research Group expanded it to offer the same abilities for Turbo Pascal and BASIC programs and for stand-alone screens that could be displayed from the DOS prompt with a supplied utility program.

For designing single text-mode screens, Saywhat is impressive. You can learn it easily and appreciate its ease of use. The drawback to Saywhat is in prototyping for flow of control. The only options are to create a program in one of the languages Saywhat works with or to create a DOS batch file that calls up one screen after another in sequence, with little opportunity to explore branching paths or go backward. And every screen is a file, no matter what program you use it with. Once again, operation is fast only if you use a RAMdisk, and even then it clutters your disk with files.

Dan Bricklin's DEMO Program

Now I have set my previous tools aside in favor of Dan Bricklin's DEMO Program, the new prototyping software by the well-known co-creator of VisiCalc. DEMO ($74.95 from Software Garden Inc., P.O. Box 238, West Newton, MA 02165; (617) 332-2240) is the outgrowth of a tool Bricklin developed originally for his own use. It goes a long way toward the tool I have been seeking—especially at the inexpensive price, which I applaud. Had DEMO been available when we began work on Wordvision, I bet we would have shipped it at least three (financially very significant) months sooner.

I am a believer in the "artistic," as opposed to "engineering," model of program design. I think good programs evolve as their structure fills and you have a chance to see what works and what doesn't. But some programmers I know prefer to work under the engineering model. ("You don't want to redesign the bridge after it is halfway built.") In particular, it seems to me programmers don't like coding interface changes. One I often work with refers to this as "shoelace tying" programming.

DEMO gives an interface designer the chance to noodle through a dozen different approaches before the programming meter starts ticking and before any expensive data or control structures get built that would be expensive to revise. Incidentally, DEMO in no way imposes a user-interface viewpoint of its own, except for the implicit one of the choices Bricklin himself has made. To illustrate this, the sample files included with DEMO show how to implement the interface for a sample task in three significantly different ways.

A Hot Program

Trip Hawkins, founder of Electronic Arts, has enunciated

(continued from page 25)

a trinity of desirable qualities for a program: hot, simple, and deep. By this standard I rate DEMO in the high percentiles for both hot and deep and a bit lower but still good on simple.

Hawkins' hot criterion can be interpreted in a couple of ways. You can take it to mean raw performance—jet-propelled accomplishment of a task at hand. DEMO certainly fits that bill; for one thing, it can jump through a series of full-screen displays as fast as you can press the keys to call them up. Hot can also mean topical—filling a need. Some work is not worth doing at all until the proper tools exist to bring the job down to reasonable proportions—many of the "what if" evaluations now done with spreadsheets probably fall into this category. "What if" experimentation with user-interface approaches, and fine tuning of them, may also have fallen in this category—until DEMO.

Now let's talk deep. Frankly, I am astonished at the depth of detail and refinement Bricklin has worked into his first release of DEMO. On occasion after occasion, as the words "Wouldn't it be nice if . . ." started to take form in my mind, DEMO provided just what I was looking for. (I suppose it helps to have beta testers of the ilk of Lotus' Mitch Kapor.)

Suppose, for example, you want to print faithful images of screen designs that make generous use of line drawing and symbols from IBM's extended character set. And suppose your printer doesn't have a character set to match. DEMO gives you a mapping feature to translate screen characters to other printer characters of your choice. You can even map individual screen characters to character sequences on the printer, using a backspace to overprint two standard characters and make up a composite that approximates to one of the special IBM set (such as a caret and vertical bar to make an up arrow).

A variation of the same feature lets you translate your screen designs into a useful form for inclusion in program code. You can direct your mapped output to a text file instead of the printer and specify a C language or Pascal language mapping, which turns characters outside the normal printable set into syntactically correct octal or hexadecimal representations. (A hex format for assembly language is curiously absent.)

The fundamentally right thing about DEMO is that it combines rich sets of tools for protyping both screen designs and flow-of-control structures and integrates them into a conceptually consistent frame of reference in which it is easy to move around. There is no perceptible border to cross in going between these two areas.

User Caring

Beyond this, DEMO possesses a caring attitude toward the user that shows throughout the program in such touches as allowing you to rearrange the order of most program lists (macros, save areas, and slides). When you are drawing lines, DEMO knows what characters to put in to make a proper junction when you draw one line to join or cross another.

There is also a familiar element from VisiCalc that contributes mightily to this program—one you might call "point and shoot." In the same way as you might build a spreadsheet formula by pointing to various cells, in effect saying "multiply this times (. . . scroll, scroll, scroll, point . . .) that," you can build a screen sequence in DEMO by flipping through the screens until you see the one you want and pressing a key to say "that one."

Two other things Bricklin does right are to make it easy to get maximum mileage out of your work by reusing it as often as is applicable and to conserve your resources with this program's analog to the "sparse matrix" approach for spreadsheet data storage (where no memory is used by cell addresses that have no real contents). The power to reuse work grows out of the Overlay features, which allow you to design only one or a few basic screens, then modify them with overlays that can have areas with both "opaque" and "transparent" areas. (In the former, the underlying display is blocked out; in the latter it shows through.)

My company's programs typically use a menu-tree structure in which soft function key legends across the bottom of the screen change as the user travels through a set of menu options. For a sequence such as this, I can paint in the basic screen and then create half a dozen overlays that pop into place in the function key display area when activated by my specified keypresses.

I don't want to give the impression that there is nothing wrong with DEMO. Refinements and extensions could keep Bricklin busy for a long time. Perhaps the major limitation at the moment is that DEMO is restricted to text mode. For people working primarily or exclusively within text mode, this is not really a limitation at all. As powerful graphics displays grow more common, however, and to the extent that windowing environments play an increasing role, it will become more important for this or a similar tool to address the graphics universe. (I sure would love to see DEMO ported to the Mac and Atari ST!)

I think DEMO is weaker than it could be in the area of screen painting, particularly attributes but also ease of putting in line and symbol characters. Macros (both built-in and from a RAM-resident key macro product) can help this, but the basic weakness remains.

Although DEMO is fairly "modeless" (that is, key combinations usually do what you expect regardless of what phase of the program you are in), there is still cleanup to be done. An example is after pressing Esc-B to bring up the Block menu, you can't move the cursor before pressing B again to begin the block.

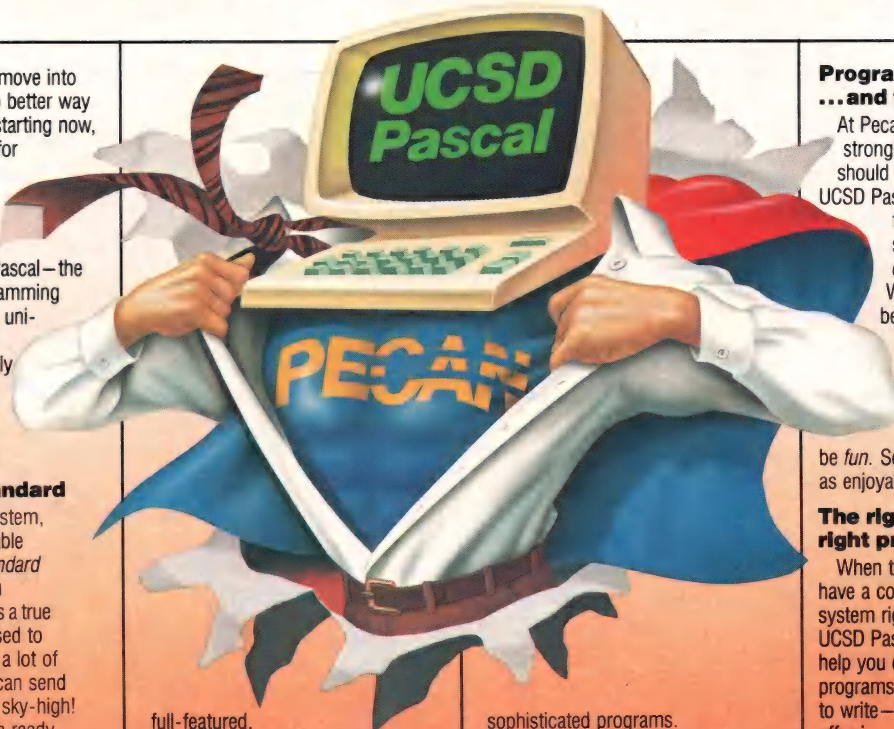
Also, the Handler capabilities (which let you specify which keystrokes or events produce which displays or actions) aren't comprehensive yet—for example, many programs use soft function key displays that change when you press the Shift (or Ctrl or Alt) key. So far, DEMO can't do handlers responding to these and other significant key actions. It also can't do much in the sound area beside beep and do simple note sequences.

Low-Rent Documentation

DEMO's documentation and packaging is self-consciously plain and low- rent. The manual is 30 pages of 8½ × 11-inch, corner-stapled and instant-printed output from Bricklin's NLQ dot-matrix printer. Its contents are orga-

Turbo, who?

Do you have to give up power and advanced potential to get ease of use and affordability? Not anymore. Because now, you can have *UCSD Pascal* for only $79.95!



If you're making your move into programming, there's no better way to go than Pascal. And starting now, you *don't* have to settle for a stripped-down version of Pascal in order to get a price that's right. Instead, you can choose UCSD Pascal—the recognized Pascal programming standard in colleges and universities throughout the country—at the incredibly low introductory price of $79.95 for your PC-DOS, MS-DOS, or other popular computer.

Start with the standard

With an *entry-level* system, you spend a lot of valuable time learning a *non-standard* form of Pascal. And you don't get all the capabilities a true Pascal system is supposed to deliver—unless you buy a lot of add-on utilities—which can send the cost of your system sky-high! Worst of all, when you're ready to tackle anything more than short, simple programs—you have no choice but to move up to a more sophisticated system (like UCSD Pascal). And at that point, you also have to *relearn* standard Pascal.

UCSD Pascal has everything you need

With UCSD Pascal, you get a full-featured, professional programming tool that's being used right now in the development of major scientific and business applications. The system comes with an outstanding text editor, a complete on-line tutorial, 8087 math coprocessor support and BCD (decimal arithmetic) *included* in the package at no extra cost. In fact, UCSD Pascal contains virtually everything you need—as standard equipment—for developing the simplest to the most sophisticated programs.

UCSD Pascal is available for MS-DOS, PC-DOS, UNIX, VMS, MSX and many other operating systems. You can use UCSD Pascal to write programs of any size on virtually any computer, and port them to any other computer. And if speed is what you're after, the latest native code version of UCSD Pascal actually benchmarks favorably with Turbo Pascal® in execution time!

Programming that's easy ...and fun!

At Pecan Software Systems, we strongly believe programming should be as *easy* as possible. UCSD Pascal was originally designed for teaching programming skills, so it's extremely *easy* to learn and to use. With UCSD Pascal, you'll be developing programs right from the start that are *easy* to write, *easy* to understand, and *easy* to maintain. We also believe that programming should be *fun*. So we've made UCSD Pascal as enjoyable to use as it is powerful.

The right tool at the right price

When the fun gets serious, you'll have a comprehensive programming system right at your fingertips with UCSD Pascal—a system that will help you develop those big-league programs you may eventually want to write—at a price you can readily afford.

Put UCSD Pascal programming power on your PC now for only $79.95! **Order by mail today or phone now 1-800-63-PECAN.** UCSD Pascal—*the* original standard of Pascal programming excellence. The new leader in Pascal price/performance.

PECAN™

The UCSD Pascal Company
Pecan Software Systems, Inc.
1410 39th Street, Brooklyn, NY 11218
718-851-3100

UCSD Pascal™

UCSD Pascal
Not copy protected
60-day money-back guarantee

Mail to: Pecan Software Systems, Inc.
1410 39th Street
Brooklyn, NY 11218
ITT Telex No. 494 8910
CompuServe Code 76703, 500

CREDIT CARD ORDERS
CALL TOLL-FREE
1-800-63-PECAN
(NYS) 1-800-45-PECAN

UCSD Pascal $79.95 (for PC-DOS, MS-DOS, AMIGA, APPLE, ATARI 520, MACINTOSH, RAINBOW, TANDY, as well as most popular 8/16/32-bit systems). **Price *includes* 8087 support and BCD.**

Please send me ____ copies of UCSD Pascal for my ____________ (Name and model of computer) which runs under ____________ (Operating system).

My disk size is 3½" ____ 5¼" ____ 8" ____.

Total amount (NYS add appropriate tax) ____________.

Payment by ☐ VISA ☐ MC ☐ US Bank Check ☐ Bank Draft

Card Number: ____________ Credit Card Expiration Date: ____/____

Name ____________

Shipping Address ____________

City ____________ State ________ Zip ________

Telephone ____________

Call or write for UNIX, VAX or other UCSD Pascal versions—and ask about our powerful Pascal add-ons, too. SCHOOLS: Contact us for our special educational discounts!

Call toll-free or enclose a check with this coupon to place an order. Please add $2.50 for shipping within the US. Foreign orders add $10 and make payment by bank draft payable in US dollars on US bank. New York State residents add appropriate sales tax.

AMIGA is a trademark of Commodore Electronics LTD. APPLE & MACINTOSH are trademarks of Apple Computer. ATARI 520 is a trademark of Atari Corporation. RAINBOW is a trademark of Digital Equipment Corporation. TANDY is a trademark of Radio Shack. Turbo Pascal is a registered trademark of Borland International

UCSD Pascal is a registered trademark of The Regents of University of California

DD

Circle **no. 200** on reader service card.

(continued from page 26)

nized to mirror the menus and submenus of the program's command structure.

This documentation approach is one that is often used, particularly in lower-priced programs. It is essentially an extension of the Kapor/Lotus-style on-screen interface that displays a set of command words in a menu and offers the ability to scroll through sentence-length explanations for the significance of each command word. The paper documentation in this scheme takes the same idea one level further; each command gets from one to several paragraphs of elaboration, noting all the dos, don'ts, and special cases. This approach works for getting the detail into a place where it is accessible for reference, but I have always found it lacking in the ability to explain how to use the program to achieve a user's goals. Bricklin is blunt about saying his program is designed, aimed, and packaged for those with the knowledge and motivation to make proper use of it. At least the documentation is clearly and simply written and seems not to miss much necessary detail.

Incongrously, whereas DEMO's packaging and positioning is aimed squarely at programmers, Bricklin seems to harbor ambitions of also selling it in the slide-show and training-system market. DEMO's capabilities certainly make it appropriate there, but users who want to use it for such purposes will have to overcome documentation not aimed at them. To accommodate customers in this market, Bricklin is considering some new pricing options. One possibility is an unlimited license to include RUNDEMO with a single slide show for a flat fee of $1,000.

A Little Taste of DEMO

To give some flavor of what DEMO can do and how it works, I used it for a small project and noted my steps as I went. Rather than building something from scratch, I experimented with possible refinements to an existing program. For fun, I picked Zoomracks, a program designed by Paul Heckel, author of *The Elements of Friendly Software Design* (New York: Warner Books, 1984) and something of a guru on the subject of user interfaces.

Zoomracks (described in *DDJ*, November 1985) uses a card-rack metaphor. A main feature of the program is its ability, when you depress one key, to switch its view (zoom) between compressed views of several information racks and a full view of a selected rack. Within a given rack, a similar feature lets you zoom between a summary top-line view of all cards in the rack and a full view of a selected card. I like these concepts but suspected the screen visuals Heckel uses to present them might benefit from some refinement. I used DEMO to test my theory.

First I loaded the Capture utility provided on the DEMO disk. This remains resident in memory and lets you import screens from other programs into DEMO. When Capture returned me to the DOS prompt, I loaded Zoomracks, manipulated it to show the first example screen I wanted to tinker with, and pressed both Shift keys at once. Capture beeped to acknowledge it had taken a snapshot of the screen. I then ran Zoomracks through the other three screens I wanted to work with (one view each way of both the card zoom and rack zoom), capturing each in the same way.

Exiting from Zoomracks to DOS, I then started DEMO itself. From DEMO's opening screen, I pressed Esc for the main command menu, I for the I/O menu, and R for Retrieve. Immediately, the first of my four screens—the multicard, multirack display—popped into view. In DEMO's terminology, this screen was my first *slide*.

The sequence Esc-R-H led me through the Run menu to the Handlers menu for my first slide. Handlers specify what DEMO should do from a given slide upon the occurrence of various events. Pressing the I key told DEMO to insert a new handler for that slide. A new menu displaying possible events popped on-screen; from it I chose pressing the F1 key as the event that would define a handler.

A menu of possible actions popped up. The V key told DEMO that I wanted to view another slide if F1 was pressed while the current slide was showing. DEMO, guessing I might want the next slide in the series, showed it to me with a small menu overlaid on one corner. My options were to press Return if that was the slide I wanted brought up, or to use F1, F2, or other search options to bring my choice into view. Because I wanted the third slide in my sequence—the multicard, single-rack view—I used F2 to bring it up and Return to select it.

Back on my starting slide, I pressed Esc-R-R (the Run option from the Run menu) to try out my new handler. The first slide showed, I pressed F1, and, zap—there was my zoomed-out view.

Ctrl-Break put me back in Edit mode, and I ran through the handler sequence again to say that, if F1 were pressed while viewing this slide, the earlier slide should be shown. I then went into the Run mode again to try it out. Success! Pressing F1 now bounced me back and forth between the zoomed-in and zoomed-out views, just like in the real Zoomracks program. After three minutes of work, I had a simulation duplicating a main feature of Zoomracks. It took me another couple of minutes set up the handlers to simulate the card zoom features of Zoomracks' F2 key.

Then I went back to the original slide to try my hand at visual redesign. I thought I would change Heckel's menu line at the bottom into a pull-down menu from the top instead. Moving the cursor to the menu line, I pressed Esc-B-B for the Begin choice from the Block menu. A one-character-size box appeared, and I used cursor keys to expand it around the whole menu line, then pressed Return to complete the selection and pressed S for save. I used this Named Save Area feature rather than just cutting and holding the menu in the normal cut/paste buffer because I would need to do another block operation to make space where I wanted the menu to go. A short series of keystrokes let me create a new save area, name it Menu, and put the menu line into it.

Using the Block feature again, I selected the top three-quarters of the screen and used the Move option to push it down a few lines, making room for the menu at the top. Then I pasted the menu into the new space. Next, a flurry of editing and attribute painting restyled the menu

THINK

FAST!

If you're a C programmer you could be a more productive C programmer.

Introducing Lightspeed C™ for the Macintosh™ from THINK Technologies, Inc.

Lightspeed C is a compiled programming environment for the Macintosh™ that gives you speed, convenience, and top quality code generation, too.

With Lightspeed C, turnaround is 1000% faster. Time to build from scratch is 3 times faster. Time to link a typical 15,000 line program is 5 seconds.[1] And generated code quality is better than any on the market.

Best of all, Lightspeed C's, integrated Edit-AutoMake-Launch environment makes turnaround a one-step process.

If you want to produce higher quality results with less time and effort, send for Lightspeed C today.

The above statements are based upon benchmarks for creating an executable version of XLISP v 1.4 (16.5K source lines) from scratch and by modifying, re-compiling, and re-linking one source file. Comparisons were performed using a 512K Macintosh with a 10MB Hyperdrive.™

	Consulair (MacC V4.0)	Aztec (V1.06G)	Megamax (V2.1)	Lightspeed (V0.40)
Generated code size (in bytes)	36770	34566	37698	33870
Program build time (in secs.)				
a. compile	887	654	354	194
b. link-to-run	99	49	95	5
c. TOTAL pgm build	986	703	449	199
Turnaround time (in secs.) (time to make a change to module xlcont.c)	159	108	127	9

☐ **Send me Lightspeed C™ fast. $175.00 for each non-copy protected compiler.**

☐ **I need more to think about, send me information about Lightspeed C.™**

Mail to:
THINK **Technologies**
420 Bedford Street
Lexington, MA 02173
Or call 617-863-5593

NAME
TITLE
COMPANY
ADDRESS
CITY STATE ZIP
TELEPHONE
☐ CHECK ENCLOSED
☐ MC ☐ VISA ☐ AMEX ACCT. # EXP. DATE
SIGNATURE

Circle **no. 234** on reader service card.

C+ The Best C Book
A Powerful C Compiler
One Great C Value $39.95

A good C book just isn't complete without a good C compiler to go with it. That's why we give you both. You get a comprehensive 450 page book and a full feature standard K&R C compiler with the Unix V7 Extensions. The Book is loaded with examples that teach you how to program in C. And our fast one pass C compiler comes with an equally fast linker so you don't waste a lot of time watching your disk drives spin. You also get a Unix compatible function library that contains more than 150 functions (C source code included). And if all that isn't enough, we offer you a 30 day money back guarantee. So what are you waiting for? The exciting world of C is just one free phone call away.

Language Features

- Data Types: char, short, int, unsigned, long, float, double
- Data Classes: auto, extern, static, register
- Typedef, Struct, Union, Bit Fields, Enumerations
- Structure Assignment, Passing/Returning Structures

Functions

abs	conbuf	feof	getcseg	isascii	movmem	replace	strcat
asm	conc	ferror	getdseg	iscntrl	open	repmem	strcmp
asmx	cos	fflush	getd	isdigit	outp	rewind	strcpy
atan	cpystr	fgets	putd	islower	peek	right$	strlen
atof	creat	fileno	getdate	isprint	perror	rindex	strncat
atoi	cursblk	filetrap	gettime	ispunct	poke	rmdir	strncmp
atol	curslin	find	geti	isspace	poscurs	scanf	strncpy
bdos	curscol	floor	puti	isupper	pow	setbuf	strsave
bdosx	cursrow	fopen	getkey	itoa	printf	setbufsiz	system
bios	cursoff	fprintf	getmode	keypress	putc	setcolor	tolower
biosx	curson	fputs	setmode	left$	putchar	setdate	toupper
calloc	delete	fread	gets	len	puts	settime	ungetc
ceil	drand	free	getw	log	putw	setjmp	ungetch
cfree	exec	freopen	heapsiz	log10	rand	setmem	unlink
chain	execl	fscanf	heaptrap	longjmp	read	sin	write
character	execv	fseek	hypot	lseek	readattr	sound	writechs
chdir	exit	ftell	index	malloc	reach	sprintf	xmembeg
chmod	exitmsg	fwrite	inp	alloc	writech	sqrt	xmemend
clearerr	exp	getc	insert	mathtrap	readdot	srand	xmemget
close	fabs	getch	iofilter	mid$	writedot	sscanf	xmemput
clrscrn	fclose	putch	isalnum	mkdir	realloc	stacksiz	xmovmem
cmpstr	fdopen	getchar	isalpha	modf	rename	str$	_exit

MIX Editor
$29.95

When you're programming in a high level language you need a high powered editor. That's why we created a programmable full/split screen text processor. It lets you split the screen horizontally or vertically and edit two files at once. You can move text back and forth between two windows. You can also create your own macro commands from an assortment of over 100 predefined commands. The editor comes configured so that it works just like Wordstar but you can change it if you prefer a different keyboard layout. The editor is a great companion to our C compiler. Because they work so well together we want you to have both. To make sure you do, we're offering the editor for just $15 when purchased with the C compiler.

NOT COPY PROTECTED

ASM Utility
$10

The ASM utility disk allows you to link object files created by Microsoft's MASM or M80 assemblers. Lots of useful assembly language functions are included as examples.

ORDERS ONLY
1-800-523-9520
IN TEXAS
1-800-622-4070

Canadian Distributor
Saraguay Software: 416-923-1500

Editor $ ________ (29.95)
C $ ________ (39.95)
C & Editor $ ________ (54.95)
ASM Utility $ ________ (10.00)
TX Residents $ ________ (6.125% sales tax)
Shipping $ ________ (see below)
Total $ ________
☐ Check ☐ Money Order
☐ MC/Visa* ________________ Exp ________
Shipping Charges: (No charge for ASM Utility)
USA: $5/Order
Canada: $10/Order
Overseas: $10/Editor • $20/C • $30/C & Editor

☐ PCDOS/MSDOS (2.0 or later)
- ☐ IBM PC Single Side
- ☐ IBM PC Double Side
- ☐ Tandy 2000
- ☐ 8 Inch
- ☐ Other ________________

☐ CPM 80 (2.2 or later)
- ☐ 8 Inch
- ☐ Kaypro II
- ☐ Kaypro 4
- ☐ Apple (Z80)
- ☐ Osborne I SD
- ☐ Osborne I DD
- ☐ Morrow MD II
- ☐ Other ________________

Name ________________________
Street ________________________
City ________________________
State ________________________
Zip ________________________
Country ________________________
Phone ________________________

MIX software
2116 E. Arapaho
Suite 363
Richardson, TX 75081
(214) 783-6001
Ask about our volume discounts.

DI

Unix is a trademark of Bell Labs. CPM is a trademark of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. WORDSTAR is a trademark of Micro Pro.

(continued from page 28)

to my taste. For the last step on the menu, I wanted to use a graphics character for visual separation between each selection. Pressing Esc-T-C (for Typing, Characters) brought up a menu displaying special characters, and a few more keystrokes selected my choice.

I needed to use the graphics character several times for the menu and found the keystroke sequence for getting one character to be excessive. The second time, before starting, I pressed Shift-F6 to turn on the Macro Learn mode. Responding to a prompt, I assigned the sequence to the Alt-1 key, then typed the usual sequence to get my graphics character. Ctrl-Break ended the macro definition. Thereafter, I could get my graphics character just by pressing Alt-1.

The final experiment I wanted to try was to give more visual impact to the zoom process. I wanted to keep some reminder of the unzoomed racks even when a selected rack was zoomed out to occupy the whole width of the screen. Using Block functions and editing, I did some surgery on the lines at the top of the multirack screen that showed the rack structure. Then I copied these lines into the paste buffer, switched the view to the single-rack screen, and pasted the new lines in place. A little more tinkering and I had the visual effect I wanted.

I gave some thought to prototyping a little animation instead of just flipping instantly from the multirack screen to the single-rack screen. It would have been easy, starting with the two slides I had and using DEMO's overlay and handler features. But I suspected the 15 minutes of work I had already done, if actually implemented for Zoomracks, would cost hours or days of coding to accomplish; so I decided to let well enough alone.

With an Esc, an I, and an S, my work so far was saved to a disk file. Included in the file were my macro for typing the graphics character and the named save area for the menu line.

If I so desired, I could then have made up a disk for Heckel with that file and with one of my 50 alloted copies of the RUNDEMO program that comes with DEMO. Although Heckel could not modify any of my prototype work without having his own copy of DEMO, RUNDEMO would allow him to view my suggestions on his computer and to run as often as he liked through all the sequences I had defined for the F1 and F2 keys.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3**.

Human Interface Design: *Jef Raskin Interview*

by *DDJ* Editors

Jef Raskin's involvement with personal computer user-interface design is hardly recent. In early issues of DDJ, *he inveighed against needlessly complex systems. At Apple Computer, he wrote many of the company's product manuals. Later he was a member of the original Macintosh design team. Today he is the founder of Information Appliance of Menlo Park, California, which currently offers a product called SwyftCard for the Apple IIe and IIc.* DDJ *will review SwyftCard next month.*

Raskin cares tremendously about user-interface design, on computers and elsewhere. During our discussion he said, "As a person I'm easily frustrated. I'm very annoyed at the little things that most people put up with." He got up and walked out of the door, shutting it behind him, and immediately opened it and reentered the room. "What a nuisance! If I had two bags of groceries and a locked door! The door is a very, very bad design! Someone will have to improve it. But we're so accustomed to it that we don't think about it."

If Raskin doesn't like doors in buildings, does that mean windows on computers are also unacceptable? In the following interview with DDJ, *he discusses a wide range of issues concerning user-interface design.*

> ***When I go to a conference and tell people why I hate a mouse I get applause.***

DDJ: For how long have you been thinking along human-interface design lines?

Raskin: When I was a computer center director at UC San Diego, there was a major computer center with a huge machine. I built another center. It didn't have fluorescent lights. It used minicomputers—the very early Data General Nova—and it was time sharing. Instead of submitting things on punch cards in the big computer center, you could come over to this little computer center I had built, and there were 32 terminals, beanbag chairs, and incandescent lighting with Japanese globes. You had individual terminals to work on. Students loved it. Even people from the big center would come over to use it.

We also had a language called FLOW, which had only six or seven commands and was very good for teaching programming. So working on better language design and the whole ergonomic question dates back to the 60s, when everyone else was looking at anything but that.

DDJ: Can you identify some of the important criteria in the design of user interfaces?

Raskin: In any user interface there's an almost unquantifiable factor of feel. And this is something I find so hard to explain. You can only achieve it by massive testing on human beings.

DDJ: Like with SwyftCard?

Raskin: SwyftCard is a little product for the Apple II, and we tested it on 1,500 people. There were focus group tests, one-on-one tests. There were small-group tests. There were tests in schools. We learned a lot. We made a lot of changes and did a lot of fine-tuning.

DDJ: But you started with something close to the product's current user interface. You had a preconception of what would work, didn't you?

Raskin: I've been concentrating on the question of how to make systems feel good for 18 years now. Sometimes that kind of concentration produces ingrown, moribund ideas, and sometimes it produces results. I think in this case it has produced results.

DDJ: Besides feel?

Raskin: There's habit, and here we're on better theoretical ground. A system should allow a person to form habits. There are two fundamental principles that help people form habits. One of them is well known in this industry, and that is making things modeless. It's just a fact of life that people can't keep track of modes. You make mode errors when you think you're doing one thing and you press a key, but because you're in the operating system and not the editor, the system does something else. I remember in the old UCSD Pascal system, I would sometimes press E for exiting, and in some other place, it meant execute—it would try to execute a letter to my mother or something. You can imagine the kinds of Pascal syntax errors you get from a letter to your mother!

DDJ: No modes. That sounds like an underlying rule.

Raskin: If a system is modeless, then you develop habits. If you want to do something, you just reach and do it that way—you don't get frustrated.

Modelessness has never been defined satisfactorily. It means that a given action has one and only one result. When you put the definition in that form, you can easily form a converse: To get a particular result, you want one and only one action. If that's the case then you don't have these branches. If you read Stewart Card's book *The Psychology of Human Computer Interaction* (Hillsdale, N.J.: Lawrence Erlbaum Associates), one of the better books in the field, you find a lot of the time you take isn't the doing time but the thinking time—working out a path. How am I going to get there from here? If you have only one way to do something, then you don't have to think about that. You can form habits even faster. If it's modeless your habits won't trip you up. In the industry we call this other property monotony, for lack of any other name. It means for a particular action, we try to have one and only one way to do it. If a system is modeless and monotonous, then you can form habits because whenever you want to do something you always do the same thing. It's like tying your shoe. Imagine if every Thursday your shoes exploded if you tied them in the usual way. This happens to us all the time with computers, and nobody thinks of complaining.

DDJ: So you have no modes, plain and simple?

Raskin: We actually did deliberately introduce one mode. We're not so doctrinaire that we won't do something when it does make sense. But our system is largely monotonous. There's generally only one way to do things. I remember reading a description of a system where the developers said, whenever they had a disagreement about how something should be done, they did it both ways. That's not a design decision.

DDJ: As in whether to use a mouse or command keys?

Raskin: Usually if you can't decide which way is better, it means you haven't found a good way to do it. On the Macintosh there's almost always a way of getting around the mouse by using the keyboard. This should have given the designers, after I left, the hint that maybe the mouse was not the right way to do it. If you're always looking for a way around it, you've obviously got some kind of problem.

DDJ: Documentation has always been a specialty of yours. Does it enter into your definition of the user interface?

Raskin: Our user manual has something interesting in it; aside from having schematics and the usual theory of operation, here's something I've never seen in any other manual that I've ever looked at—the user-interface theory of operation, or how we designed the user interface.

DDJ: You believe the user must understand the theory?

Raskin: No, I don't feel the user needs to know. I know, as a reader of manuals, I often wonder: Why do they do this? How did this come about? You can use the system without reading any of the theories of operation. You can learn the system by reading about 40 pages. Some people learn it from a reference card. We have only five commands. We could have had a manual that just ex-

(continued from page 33)

plained the five commands. But this manual is part of our user interface.

Let's talk about manuals. We have a long, cross-referenced, handmade index. A lot of work went into this. Before we did this we typeset the manual for testing because we wanted people to feel they had a finished product. Then we got feedback not only on the product but also on the manual, so what people get as our first-edition manual is a tested manual. Manuals are an integral part of the product, not an afterthought. This whole product was designed from the very beginning with the manual in mind. In fact, the very first thing I did when I started the company was write my dream manual for this product. Then I built it and finally wrote a real manual.

DDJ: That brings to mind Apple's ad that boasts of the minute amount of documentation needed to learn the Mac compared to an IBM computer.

Raskin: Our first idea was to have a thin manual. We tried that—experts said, "Wonderful," but beginners said, "I don't know what I'm doing." If you say that an insert command undoes the latest block delete, people may have forgotten what a block delete is and *insert* is a funny word. We wrote the manual so we explained everything. If we had to explain something five times, we explained it five times. Don't get lazy in the manual!

DDJ: Beyond modelessness and monotony, what else do you have in mind as you develop a product?

Raskin: One of the most important concepts is that things you do frequently must be fast. Things that you do infrequently can be slower. People have critiqued our way of setting the widths of paragraphs, saying it seems a little baroque, but you hardly ever do it. One thing you do very often is move the cursor, and one of the big advances of this product is the cursor-moving mechanism. This was not theoretical. It hit me while driving through Marin with my wife. No amount of theoretical analysis will give you a better system without inspiration. I know of no way of automating that process.

DDJ: But you began with an antipathy toward the mouse?

Raskin: I happen to hate mice, and I have since 1973 when I first started using them at Xerox PARC. I always preferred joysticks. I did want a different cursor-moving mechanism on the Macintosh. That was designed as a graphic machine, and you do need a graphic cursor-moving device. SwyftCard is nongraphic and doesn't need a graphic device. *(At this point Raskin started to demonstrate the SwyftCard while he talked.)*

First of all, it does not require you to move your fingers from home row, and it uses your thumbs, which are under-utilized fingers. Even in touch-typing the right thumb is used for the space bar only; the left thumb does nothing. Strangely enough, the Dvorak keyboard, for all its supposed efficiencies, doesn't use the thumbs either.

To get anywhere in about 20 pages of text, you have to type an average of 3½ characters. Research has shown that with cursor-moving keys the average time is around ten seconds, and with the mouse it drops to around four seconds. Here it drops to around a second. So it's somewhere between two and four times faster than a mouse, and it's a heck of a lot less expensive to manufacture.

DDJ: Had you seen anything like this that helped you in the early stages of development?

Raskin: No, after I designed this I learned about the Find command in EMAX, which is also an incremental search. It has a few problems: First of all, you have to go into Find mode; and second, it ends up on the last character of the pattern, which is a big mistake. The other thing with EMAX is, when you leap somewhere and you start typing, you're still adding into the pattern. So it's modal, and it puts you on the last character. It's only one direction.

Let me make a typical error. I want to move the cursor to the word *good*, so I should press the left Leap key and type "good." I'll press the right

The FASTEST, EASIEST, MOST POWERFUL BASIC EVER!

Amazing New Advance – ments for an Old Friend.

ZBasic is an incredibly advanced and powerful BASIC—but—it's still the old BASIC you're used to. Instead of spending 6 months of your life learning another complicated language, let ZBasic put your programs into light-speed, now! (If you know BASIC, you know ZBasic.)

How Fast is ZBasic?

Lightening fast. Four years of intense development have produced the ultimate BASIC. ZBasic is "Compiled BASIC," and generates stand-alone applications that make any other BASIC completely obsolete. Just look at these speed comparisons.

Sieve Benchmark on Different PC's			
Macintosh™		**Apple IIe, IIc**	
ZBasic™	7.4 sec.	ZBasic™	486 sec.
Mbasic™	684 sec.	Applesoft™	5,401 sec.
IBM® PC (8088)		**Z-80** (CP/M™-80, TRS-80™)	
ZBasic™	13.7 sec.	ZBasic™	30 sec.
BASICA™	2,190 sec.	Mbasic™	2,520 sec.

10 iterations of the Sieve from Byte, January, 1983

Compiler Speed/Interpreter Ease.

Like a BASIC interpreter, ZBasic allows you to write and execute your programs immediately! No messy "Linkers," "Loaders," or clumsy "Subroutine Packages" like most other compilers. To compile and edit, simply type "RUN." Debugging works the same as the interpreter, too. Just type "BREAK" or "CTRL C" to get back to the editor.

Lightning-Fast Compilation.

Computer Language Magazine says. "Compilation is amazingly fast…" After typing "RUN," ZBasic compiles your program at blinding speed—40 lines per second.

Works the Same on All Computers.

If you're tired of throwing away your old programs everytime you switch to a new computer, ZBasic is for you. Source code is portable from one computer to another, and since ZBasic uses Device Independent Graphics and Disk File commands, your programs automatically "Adapt" to any other computer. And the ZBASIC editor is the same on all versions—regardless of the computer.

Einstein Math.

ZBasic offers programmers a math package that surpasses anything else in the industry! (Yes, ZBasic is even better than FORTRAN, PASCAL, MODULA-2 or any other language available!) You will have up to 54 digits of user-selectable accuracy at your power.

"Superb Documentation!"

"The 387 page ZBasic manual is a model of clarity and organization. The documentation is superb, solidifying our impression that someone worked incredibly hard to make ZBasic a benchmark for all other BASIC Compilers."
PC WEEK, Nov. 12, 1985

Easy Structure—If You Want It.

ZBasic helps you "Structure" your programs in a way that's easy and simple…you may use GOSUB or GOTO with names or line numbers. Supports multi-line LONG IFs and LONG FNs. LIST programs with—or without—line numbers! ZBasic automatically indents loops and structures in LISTings, too.

ZBasic Users Say:

"Awesome! It's about time! Great! Unbelievable!"
J.R. CPA
Seymour, MO

"....fast, generates stand alone programs, requires only modest amounts of memory, has outstanding compilation speeds and...was bug free and felt solid. And the price is very attractive."
Bruce W. Tonkin
COMPUTER LANGUAGE

"ZBasic is a powerful offering for BASIC programmers. It provides the flexibility of Turbo Pascal and the speed of compiled BASIC, all at a price that can't be matched. Kudos to Zedcor and to all users who make wise decisions to use ZBasic to the fullest."
Garry Ray
PC WEEK

"The best I have ever seen. I love it! You should be proud of this product."
R. R. Manager
Mesa, AZ

"Mind-blower! Easily the best BASIC I've ever seen!"
J. D.
Baltimore, MD

Versions shipping now!

- Macintosh , Apple IIe - IIc (128K & DOS 3.3)
- IBM PC and MSDOS 2.1 & Compatibles
- Kaypro Graphics version (CP/M-80)
- CP/M-80 2.0 or greater (Z80 only)
- TRS-80 Model 1/3, Model 4/4p

Customized for YOUR Computer:

MSDOS™ and Compatibles: Including PC, XT, AT, jr., Tandy™ 500-1200-2000-3000 and all Compaqs™. Creates fast stand-alone .COM files. Supports a mouse, highlights keywords and lots more.

Macintosh™: Complete Toolbox ROM calls support, creates 68000 Native Code, Macintalk and Appletalk support, program size to 4 megabyte, math accuracy from 8 to 240 digits. Incredible program speeds!

Apple™ IIe, IIc: Mouse support for both the IIe and IIc, Super Hi-Res graphics support (560x192 and 280x192 and lo-RES support tool). Advanced Graphics commands like CIRCLE, BOX, FILL etc. **and** you can Mix Graphics and Text on the screen like a PC. DOS 3.3 support (PRODOS coming this summer) Requires Apple //e or //c with 128K but programs created with ZBasic™ will run on a 64K Apple II+.

Z80™ Machines: CP/M™-80 2.0+, TRS-80™ model 1, 3 or 4 and a special graphic version for Kaypro CP/M.

ONLY ZBASIC GIVES YOU THESE FEATURES:

- **Highlights errors…makes debugging easy!**
- **Not Copy Protected**
- **Never any Royalties or Runtime fees for programs you sell.**
- **Direct commands (Speeds logic testing like an interpreter)**
- **Super Single-Step debug**
- **CHAIN with variable passing. (Share all or some variables)**
- **Create transportable subroutines and functions**
- **Multi-line LONG IF. Multi-line LONG FNs (argument passing)**
- **Decimal, HEX, OCTal or BINary support.**
- **Device-independent Graphics and File I/O.**
- **Never does String "Garbage Collection"**
- **Comes with "Quick" and "Shell" sort source code**
- **Built in "HELP" screens lets you get answers fast.**
- **Long variable name (15 characters)**
- **Loops: WHILE-WEND, DO-UNTIL, FOR-NEXT-STEP**
- **Serial Port and Modem support**
- **Easily load your old BASIC programs saved in ASCII**

$ 89.95

Send me ZBasic right away! $89.95 complete.
plus shipping

CREDIT CARD—MASTERCARD/VISA/AMEX/C.O.D.

Card Expiration Date: ____________

My computer is a ____________

Name

Address APT. #

City State Zip

Day Phone

MAIL TO: ZEDCOR, INC.
4500 E. Speedway, # 93
Tucson, AZ 85712

ZBasic™

Arizona Residents add 5% Sales Tax. In U.S. add $5.00 shipping. C.O.D. add $5.00 per order. Outside U.S. and Canada, add $20.00 per shipment for postage (U.S. currency only.)
C, Turbo Pascal, IBM PC, Fortran, Pascal, MODULA COBOL are trademarks of their respective companies—Macintosh is a trademark of Macintosh Laboratories, Inc.

30 day money-back guarantee.

ORDER TOLL-FREE:
1-800-482-4567 Technical Support: (602) 795-3996

Circle **no. 160** on reader service card.

(continued from page 34)

Leap key and type. It found it anyway. The system does one thing that all systems should have done from day 1: If you tell it to search one way for something and it doesn't find it, it searches the other way in case you made a mistake. Most systems didn't do this because if you did find it then you've lost your place. In this system if you want to go back, you just bang on the keyboard. *(Raskin slams both hands on the keyboard, and the cursor returns to the point in the document at which his search began.)*

The other thing about this system that allows us to use this paradigm is that the longest search and display through all the text takes 300 miliseconds. Cursor motion is very important, and we've got it better than anyone. Speed is very important, and here on a stupid old 6502 running at 1 megahertz, we're moving the screen and doing things faster than you will see on any computer from any manufacturer at any price with anybody's software.

DDJ: On the Mac development team, was there an absence of theoretical drive?
Raskin: The original team was Bud Tribble and myself for software development. When Steve took over, Bud Tribble left. And I left. He then brought in a group of people who essentially implemented a standard operating system and just put a different appearance on the front of it. There was a strong theoretical drive initially; all the people who were doing that left. Our name for the word-processing program you get with the Macintosh is Macwait. If a little clock ever appears on a computer of mine, I'll shoot it. A computer is supposed to be fast.

DDJ: Could you talk about the cursor design on SwyftCard?
Raskin: One of the things we've done is we've paid a lot of attention to details people have taken for granted for years. The way I've gotten out of my ruts is by watching people try to learn to use our own systems. The cursor is in two parts. There is the blinking part, which we call the cursor, and the other part, which we call the highlight. The rule is, when you type a character, it will always appear where a blinking cursor is, and whatever character was underneath it gets moved out of its way. Always. It will never happen to the right or left of the cursor. The highlight is where the next character to be deleted will be deleted when you press the Del key. When you're typing it shows exactly what's happening. I don't know how many times I've made the mistake of mispositioning a cursor on a system that deletes to the left and inserts to the right, like on the Macintosh. That causes a lot of confusion.

We spent six months before we realized a cursor has to be in two parts. We played with cursors between the letters, on the letters, cursors that flickered back and forth—dozens of designs. You always want to use a left delete after you've been typing. If you move the cursor somewhere, you always think of words from the beginning of the words. You move there, and you always want to delete the other way after you've moved the cursor. We observed that to be a 99 percent phenomenon. So we have it automatically. Whenever you leap, the cursor knows to delete to the right, and whenever you're typing, it automatically backspaces. This is definitely modal, and once in a blue moon you will make a mode error. You will press Delete and take out the wrong character. But it's so much gain.

DDJ: So you've eliminated many commands normally found in an application, especially in a program with several applications.
Raskin: Throwing out commands is a very big thing. This system does word processing, information retrieval, telecommunications, and calculations, and it has five commands. Any other word processor has more than five.

DDJ: You've complained that the Mac ended up with a traditional operating system. How does your system differ?
Raskin: We threw out the whole concept of an operating system. By definition, an operating system is the program you have to fight with before you can fight with an application. For a single-user system on which you're not developing software, who needs an operating system? There is no operating system running underneath this. The editor runs right on the bedrock of the silicon. There are no menus. You know that people like broad, flat menus rather than deep menus. What is the broadest and flatest menu you can imagine? A few things labeled on the fronts of keys. Everything is available instantaneously and simultaneously. If I want to look up a telephone number, I don't have to get into Information Retrieval mode, I just use the Leap command. Leap is information retrieval. If I want to do a calculation, do I say "Calculation mode" or pull down the calculator? No, I simply type the equation and press the Calc button. That's the way it should always have been.

DDJ: The most basic concept to most interfaces is the separation between applications.
Raskin: In today's world! In tomorrow's world, interfaces like this will clobber the usual present kind, and in a few years many kinds of products will have this kind of interface.

DDJ: What's held people up from seeing this sort of integration?
Raskin: We already have integrated software, and that's integration by a menu. We call this homogenized. Why didn't I see this years ago? I have no idea.

DDJ: Can any application be homogenized?
Raskin: Yes. You name it, and I can homogenize it. The basic principle can be applied to all applications that I know of on computers. The work we've done here can be applied to any computer, small or large, and any application.

DDJ: Given the right hardware?
Raskin: Given the right software. Most people think the Apple II is the wrong hardware for any spiffy human interface. Part of the problem is this doesn't show up on television. All you see is your work. But isn't that what you want to see? Do you want to see the computer mechanism, or do you want to see your work? We don't waste any of the

SAS Institute Inc. Announces

Lattice C Compilers for Your IBM Mainframe

Two years ago...

SAS Institute launched an effort to develop a subset of the SAS® Software System for the IBM Personal Computer. After careful study, we agreed that C was the programming language of choice. And that the Lattice® C compiler offered the quality, speed, and efficiency we needed.

One year ago...

Development had progressed so well that we expanded our efforts to include the entire SAS System on a PC, written in C. And to insure that the language, syntax, and commands would be identical across all operating systems, we decided that all future versions of the SAS System—regardless of hardware—would be derived from the same source code written in C. That meant that we needed a C compiler for IBM 370 mainframes. And it had to be good, since all our software products would depend on it.

So we approached Lattice, Inc. and asked if we could implement a version of the Lattice C compiler for IBM mainframes. With Lattice, Inc.'s agreement, development began and progressed rapidly.

Today...

Our efforts are complete—we have a first-rate IBM 370 C compiler. And we are pleased to offer this development tool to you. Now you can write in a single language that is source code compatible with your IBM mainframe and your IBM PC. We have faithfully implemented not only the language, but also the supporting library and environment.

Features of the Lattice C compiler for the 370 include:

- **Generation of reentrant object code.** Reentrancy allows many users to share the same code. Reentrancy is not an easy feature to achieve on the 370, especially if you use non-constant external variables, but we did it.
- **Optimization of the generated code.** We know the 370 instruction set and the various 370 operating environments. We have over 100 staff years of assembler language systems experience on our development team.
- **Generated code executable in both 24-bit and 31-bit addressing modes.** You can run compiled programs above the 16 megabyte line in MVS/XA.
- **Generated code identical for OS and CMS operating systems.** You can move modules between MVS and CMS without even recompiling.
- **Complete libraries.** We have implemented all the library routines described by Kernighan and Ritchie (the informal C standard), and all the library routines supported by Lattice (except operating system dependent routines), plus extensions for dealing with 370 operating environments directly. Especially significant is our byte-addressable Unix®-style I/O access method.
- **Built-in functions.** Many of the traditional string handling functions are available as built-in functions, generating in-line machine code rather than function calls. Your call to move a string can result in just one MVC instruction rather than a function call and a loop.

In addition to mainframe software development, you can also use our new cross-compiler to develop PC software on your IBM mainframe. With our cross-compiler, you can compile Lattice C programs on your mainframe and generate object code ready to download to your PC.

With the cross-compiler, we also offer PLINK86™ and PLIB86™ by Phoenix Software Associates Ltd. The Phoenix link-editor and library management facility can bind several compiled programs on the mainframe and download immediately executable modules to your PC.

Tomorrow...

We believe that the C language offers the SAS System the path to true portability and maintainability. And we believe that other companies will make similar strategic decisions about C. Already, C is taught in most college computer science curriculums, and is replacing older languages in many. And almost every computer introduced to the market now has a C compiler.

C, the language of choice...

C supports structured programming with superior control features for conditionals, iteration, and case selection. C is good for data structures, with its elegant implementation of structures and pointers. C is conducive to portable coding. It is simple to adjust for the size differences of data elements on different machines.

Continuous support...

At SAS Institute, we support all our products. You license them annually; we support them continuously. You get updates at no additional charge. We have a continuing commitment to make our compiler better and better. We have the ultimate incentive—all our software products depend on it.

For more information...

Complete and mail the coupon today. Because we've got the development tool for your tomorrow.

SAS®

SAS Institute Inc.
SAS Circle, Box 8000
Cary, NC 27511-8000
Telephone (919) 467-8000 x 7000

I want to learn more about:

☐ the C compiler for MVS software developers
☐ the C compiler for CMS software developers
☐ the cross-compiler with PLINK86 and PLIB86

today...so I'll be ready for tomorrow.

Please complete or attach your business card.

Name ______________________

Title ______________________

Company ______________________

Address ______________________

City ______________ State ________ ZIP ________

Telephone ______________________

Mail to: SAS Institute Inc., Attn: CC, SAS Circle, Box 8000, Cary, NC, USA. 27511-8000. Telephone (919) 467-8000, x 7000

4/86 DDJ

(continued from page 36)

screen with indicators or messages. It's all yours. On a 64K Apple II with our system, the user gets 40K. If you buy a 128K Mac and MacWrite, you get 20K out of 128K. Our entire code for this product is 14.5K. Nobody even thinks you can write an application, much less four or five applications, in under 16K bytes. Those numbers have not been heard for years. Go back to the early days of *DDJ*!

DDJ: But you must feel some frustration with this product on the Apple II?

Raskin: Sure. The keyboard layout isn't optimal. The Leap keys should be larger. They weren't designed for leaping use. There are many applications on other hardware that we simply don't do here. For one thing, we don't have the spreadsheet integrated because the Apple finally ran out of power at a certain point. But the hardware is not so important. And the amount of stuff you can do with an 8-bit processor and even 16K bytes of memory have not been exhausted by humanity and will never be.

People like Steve Jobs say that the way to get simplicity is greater complexity. Well, they don't put it quite that boldly. What they say is, if we have intelligent enough systems and big enough systems, we can make them very easy to use. I have this stupid idea of simplicity via simplicity—and for many applications it works. I'm not denigrating all of AI and that stuff. I think there's a lot to be gained from it but not for little things that people are going to use on an everyday basis.

I was a visiting scholar in the artificial intelligence lab at Stanford, and I had an office at PARC. I was never an employee at PARC.

DDJ: So you think the mouse created a barrier toward designing better user interfaces. What's better for graphics?

Raskin: My favorite graphic input device is a tablet. That's what I find easiest to use. It feels like a pencil. I spent years practicing, using pencils. I've thought the mouse was a mistake for a decade and a half now. I think it's being foisted on a lot of people. When I go to a conference and I tell people why I hate a mouse, I usually get applause.

DDJ: Specific applications create certain needs in interface design. What's an example of this?

Raskin: Most people who do a lot of heavy telecommunications end up having two computers because, if you want to receive messages at an arbitrary time, you have to leave your telecom package up. With your telecom package up, you can't do anything else. On this system, it's always in Telecom mode. If you send me a message, whether I'm working or not, I won't be interrupted. If I'm not there, it'll just accept it; if I am there, I can finish my thought and then read your message. So here's a place where modelessness buys you a whole computer.

DDJ: Have the main issues changed in terms of what a programmer wants and a user needs?

Raskin: I think so. There are some programmers I haven't hired here because they say, "Hey I'm not going to get a chance to hack at systems or Unix or anything like that here." The programmers we do have here have as their first priority making things work well with people. That means you can't have an interest in developing a new and better operating system. In a few years all computer-science departments will have user-interface courses; some do now. And that will become more and more recognized as legitimate and worthy. We have to have a stronger theory of human interfaces. Part of what I'm doing is developing such a theory in my spare time.

DDJ: What would you concentrate on in a user-interface class?

Raskin: First of all, you've got to get people to recognize when they're having trouble. People always say, "It's me," but it's the computer design that's so dumb. So first, you have to get programmers out of the cocky position of believing they know how to design stuff and to a humble position that if a person is having a problem, it's not the person who's dumb, it's a dumb design. That's the start. And have them learn statistics and experimental design. And then understanding how human beings learn and work—cognitive psychology and learning theory.

DDJ: Will different languages help in the creation of better interfaces?

Raskin: Definitely. We used Forth. That helped us here because it was small and runs like a bat out of hell. You don't have many compunctions about dropping down to assembly language if you really need speed. So for sheer performance, it was a good choice. For human-interface design, most of the languages like PROLOG and LISP or APL are totally inappropriate. I've been reading some articles about people writing simulators on which you can test human design interfaces. They say it's very slow, which invalidates it. Unless they can simulate the real speed, they're not getting any useful data. There's no language per se.

DDJ: What thoughts did you have along these lines while you were doing things for *DDJ*?

Raskin: Go back and read the very first article I ever published in 1976 in *DDJ*. Read Jim Warren's little comment about me at the bottom. *["Jef Raskin is well known for his heretical belief that people are more important than computers and that computer systems should be designed to alleviate human frailties, rather than have the human succumb to the needs of the machine."]* It's still true, word for word. I've been trying to fulfill that belief ever since.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4**.

NEW RELEASE

Ecosoft's Eco-C88 Rel. 3.0 C Compiler

$59.95

Release 3.0 has new features at an unbelievably low price. ECO-C88 now has:

- Prototyping (the new type-checking enhancement)
- enum and void data types
- structure passing and assignment
- All operators and data types (except bit fields)
- A standard library with more than 200 functions (many of which are System V compatible for greater code portability)
- cc and mini-make that all but automates the compile process
- 8087 support (we sense the 8087 at runtime – no dual libraries)
- ASM or OBJ output for use with MSDOS linker
- Tiered error messages – enable-disable lint-like error checking
- Fast compiles and executing code
- Expanded user's manual
- Enhanced CED program editor (limited time offer)

We also offer the following support products for Eco-C88.

CED Program Editor $29.95

CED now supports on-line function help.
If you've forgotten how to use a standard library function, just type in the name of the function and CED gives you a brief summary, including function arguments. CED is a full screen editor with auto-flagging of source code errors, multiple windows, macros, and is fully configurable to suit your needs. You can edit, compile, link, and execute DOS commands from within the editor. Perfect for use with Eco-C88. For IBM PC, AT and look alikes.

C Source for Standard Library $10

Contains all of the source code for the library functions that are distributed with Eco-C88, excluding the transcendentals and functions written in assembler.

($20 if not with order)

Developer's Library $25

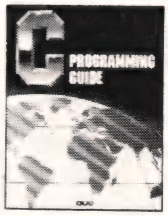
Contains the source code for all library functions, including the transcendentals and those written in assembler. Perfect for the developer that wish to write their own custom functions or learn how we implemented the Eco-C88 library.

($50 if not with order)

ISAM Library $15

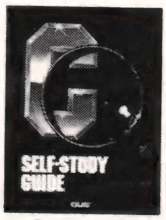
Contains the code from the **C Programmer's Library** in relocatable format (i.e., .OBJ) including the delete code for the ISAM file handler.

($30 if not with order)



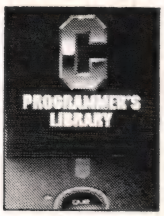
C Programming Guide $20

After reading the 1st edition, Jerry Pournelle (**BYTE** Magazine) said: "I recommend this book . . . Read it *before* trying to tackle Kernighan and Ritchie." The second editon expands this best seller and walks you through the C language in an easy-to-understand manner. Many of the error messages include references to this book making it a perfect companion to Eco-C88 for those just starting out with C.



C Self-Study Guide $17

(Purdum, Que Corp.). Designed for those learning C on their own. The book is filled with questions-answers designed to illustrate many of the tips, traps, and techniques of the C language. Although written to complement the Guide, it may be used with any introductory text on C.



C Programmer's Library $20

(Purdum, Leslie, Stegemoller, Que Corp.). This best seller is an intermediate text designed to teach you how to write library functions in a generalized fashion. The book covers many advanced C topics and contains many useful additions to your library including a complete ISAM file handler.

Eco-C88 C compiler requires an IBM PC, XT, or AT (or compatible) with 256K of memory, 2 disk drives and MSDOS 2.1 or later. Call today:

1-800-952-0472 (for orders)
or
1-317-255-6476 (tech. info.)

ECOSOFT INC.

Ecosoft, Inc.
6413 N. College Ave. • Indianapolis, IN 46220

AMERICAN EXPRESS MasterCard VISA

TRADEMARKS: ECO-C88, ECOSOFT

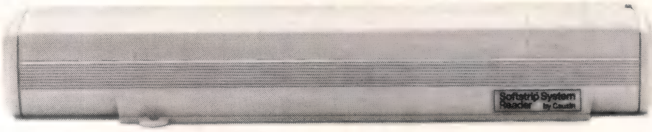
Circle **no. 89** on reader service card.

It's amazing what you can reveal when you strip.

Introducing a shape that's about to turn on an entire industry.

The Softstrip™ data strip. From Cauzin.

This new technology allows text, graphics, and data to be encoded on a strip of paper, then easily entered into your computer using a scanning device called the Cauzin Softstrip™ System Reader.



The Cauzin Softstrip System Reader replaces tedious typing by scanning the strip and reading it into your computer.

Creating a simple, reliable and cost efficient way to distribute and retrieve information.

Softstrip data strips, like those you see here, can contain anything that can be put on magnetic disks.

Facts. Figures. Software programs.

Video games. Product demonstrations.

Sheet music.

The Cauzin Softstrip System Reader is now compatible with the IBM PC, Apple II and Macintosh.

A single strip can hold up to 5500 bytes of encoded data.

It can stand up to wrinkles, scratches, ink marks, even coffee stains.

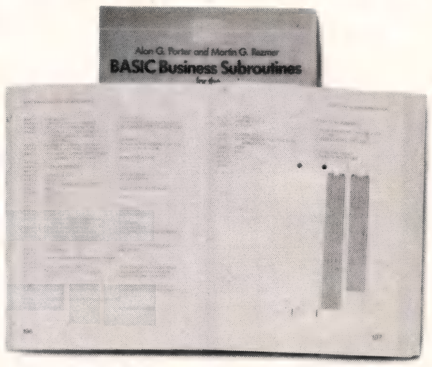
And it can be entered into your computer with a higher degree of reliability than most magnetic media.

Simply by plugging the Cauzin Reader into your serial or cassette port and placing it over the strip.

The reader scans the strip, converts it to computer code, and feeds it into any standard communication interface.

Because strips are so easy to generate, most of your favorite magazines and books will soon be using them in addition to long lists of program code.

And you'll be able to enter programs without typing a single line.



Soon everyone will be stripping as data strips appear in popular magazines, computer books and text books.

There is also software for you to generate your own strips. Letting you send everything from correspondence to business information using our new technology.

Find out how much you can reveal by stripping. Just take this ad to your computer dealer for a demonstration of the Cauzin Softstrip System Reader.

Or for more information and the name of the dealer nearest you, call Cauzin at 1-800-533-7323. In Connecticut, call 573-0150.

Softstrip™
COMPUTER READABLE PRINT

Cauzin Systems, Inc.
835 South Main St., Waterbury, CT 06706

Circle **no. 226** on reader service card.

MAKING THE GOOD LIFE EVEN BETTER

Someone once said that there is nothing new under the sun. Wouldn't life be boring if that were indeed true? The data strips on the right contain the program described in the article "The Game of Life in Expert-2", by Jack Park, which appears in this issue. It's a prime example of how something, in this case the game of **LIFE** itself, can, indeed be improved.

The game of **LIFE** was invented years ago by John Horton Conway. Over the years, the game has evolved into a popular cerebral exercise for programmers and math majors alike. At first the game was played on graph paper, but the advent of modern technology moved it to the computer which plays the game thousands of times faster. Now millions of computer enthusiasts are captivated by this devilishly simple, yet marvelously complex quintessential computer diversion.

The rules of the game are quite simple. Imagine that you have an infinite grid of squares, each one being either alive (on) or dead (off). Each square (called a "cell") lives or dies into the next cycle (called a "generation") based on its current state and that of its neighbors. The grid of cells is represented by a graphic display on your computer screen. After setting up an initial configuration of living and dead cells, you start the simulation. The patterns will change on the screen as cells live and die.

Mr. Park's improvement on the theme is interesting because of his approach. Instead of writing a traditional program for the simulation, he has created an array of intelligent cells using an inference engine written in Expert-2, a superset of FORTH.

Read in the data strips, following the directions that came with your Cauzin reader. You'll need the Expert-2 programming environment to operate this program. Refer to Mr. Park's article in this issue for operating instructions.

StripWare Library No. 202

1 2 3

Softstrip

Simple Plots with the Enhanced Graphics Adapter

by Nabajyoti Barkakati

I needed a module that could join two points on the screen.

Recently I needed to generate some engineering graphs on an IBM Enhanced Color Display with the Enhanced Graphics Adapter (EGA). I was using a program written in Lattice C Version 2.14 to do the computations, and all I needed was the capability to draw a simple x-y plot, taking advantage of the EGA's 640-by-350 pixel high-resolution mode. An obvious solution would have been to use IBM's Graphics Development Toolkit, but it seemed to be the wrong tool for the job I had in mind.

I assembled my resources. The ROM BIOS video functions (interrupt *10h*) came to mind; a quick search of the Lattice C reference manual produced the *int86( )* utility function. Armed with these tools, I wrote a set of graphics functions to do my job. Here I present the most interesting of these modules, which set the display in high-resolution graphics mode and perform line drawings.

The EGA can be set up to operate at its highest resolution mode by using the mode number *16* in the ROM BIOS video interrupt. In its basic configuration with 64K RAM, this mode allows four colors. The 640-by-200 pixel resolution of the Color Graphics Adaptor (CGA) can be emulated by using mode number *14*.

The BIOS provides a function that allows you to write a pixel; I needed a module that could join two points on the screen. This meant that I had to brighten up a lot of pixels that are close to an imaginary line drawn between the two end points. A tool to help me do this was close at hand—in *DDJ*, in fact.

The May 1985 issue (#103) contained the article "Using Decision Variables in Graphics Primitives" by Tom Hogan, in which the author generalized Bresenham's decision variable method. Bresenham's original algorithm for line drawing is described in the book *Fundamentals of Interactive Computer Graphics* by James D. Foley and Andries Van Dam (Reading, Mass.: Addison-Wesley, 1982). The algorithm in the book works for lines with slopes between 0 and 1, but can be easily modified to work with lines of arbitrary slope. I have made such modifications in my line drawing function. (See *v_draw* in Listing Two, page 74.) Note that I have chosen to represent the lower left corner of the screen as the pixel (0,0), though the upper left corner is considered to be the origin in IBM's ROM BIOS.

In Listing One, page 74, I show a sample program that plots sin(x) vs. x on the screen. It's a simple demonstration of the use of graphics primitives. A better way to use the capability would be to write a module that would nicely map the arrays y and x of size n onto the screen. It would first find the true maxima and minima of the data, pick a nicer set of numbers for these extremes, then scale each point onto screen coordinates and use *v_draw* to draw linear segments on the screen to make the plot. A coordinate grid under the plot would make the output even more attractive. But all this follows naturally once you can connect any two points on the screen, which is what the code presented here permits. I'm sure you wouldn't want me to take away all the fun of setting up your own customized plotting package.

Bresenham's decision variable method, as Hogan explained in his article, can be used to generate any well-behaved curve of the form G(x,y)=0. *Hogan's implementation of the algorithm was intended for medium-resolution graphics; Barkakati's interest is with the considerably higher resolution EGA affords.*

What Bresenham's method gains you is chiefly speed, an important consideration when doing high-resolution curve drawing. Rather than computing square roots for every point, it depends heavily on integer math. It also creates curved line segments without gaps, a feature that leads to smoother-looking curves. Based on the value of the slope of the tangent line and the direction of its change, the method decides where, in relation to the last point, the next point goes. The slope doesn't have to be calculated for each point; key inflection points merely need to be identified. The slope and how it is changing can be used to select from all possible next points a pair of candidate points and to choose between these—the decision variable part of the approach. At any given location only two next points are reasonable, and the method finds them and chooses between them.—ed.

DDJ

(Listing begins on page 74.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5**.

Nabajyoti Barkakati, 2005 Aventurine Way, Silver Spring MD 20904

Now dBASE is bilingual.

Announcing a second language for dBASE®

C.

Now you can add richer, faster features to the dBASE you know and love with "dBASE Tools for C™"

So you can continue to program in the dBASE programming language, and yet have state-of-the-art calc speed and unique fast-painting graphics.

Here's your tool kit:

A basic engine that links C, special C libraries, and your own C functions to dBASE applications. (It supports Lattice® C, Microsoft® C, and Manx Aztec™ C.)

Arrays management and a C library of financial, mathematical, and statistical functions come with the Programmer's Library.

And the Graphics Library includes interactive business graphics like bar graphs, pie charts, exploded pie charts, marked point graphs, line charts, and XY charts.

To order, for the name of your nearest dealer, or for more information, call the Ashton-Tate Publishing Group at 800-437-4329, Ext. 242.

Sure you need C to be on the leading edge.

But you don't have to give up dBASE to get it.

Circle no. 242 on reader service card.

A 68000 Cross Assembler—Part 2

by Brian R. Anderson

In Modula-2, the definition module is like a contrast. It says what tasks will be performed without saying anything about the methods.

Last month I introduced a Modula-2 cross assembler for the Motorola 68000 16/32-bit microprocessor. In that first installment, I looked at the overall structure of the program from a data flow perspective and presented all definition modules as well as the main program module.

This month, I will discuss some of the algorithms used and present all of the implementation modules along with a stand-alone program to initialize the mnemonic lookup table.

Although this project was developed on a Z80 system using the Modula-2 System for Z80 CP/M from Hochstrasser Computing AG, the source code should compile and run on most other small-computer implementations of Modula-2. I will try to identify those areas that may be impediments to portability. The only true machine dependency is because of the assumption that INTEGERs, CARDINALs, and BITSETs all occupy 16 bits in memory. Although not all Modula-2 libraries are equivalent, I have tried to restrict myself to library routines that are common to both the Hochstrasser and the popular Volition Systems compilers. That should allow a fair degree of portability across many different compiler systems.

In Modula-2, the definition module is like a contract. It tells the users of the module (that is, programmers) what tasks will be performed without saying anything about the methods used. Definition modules are compiled separately and provide the only interface to other modules or to the main program module. The definition modules presented last month indicated the tasks that are to be performed during the execution of X68000; the implementations presented this month describe the algorithms used to accomplish those tasks.

Brian Anderson, 2977 East 56th Ave., Vancouver, B.C. V5S 2a2 Canada

LongNumbers

The data type *LONG* (an array of integers) simulates 32-bit hexadecimal numbers; the implementation module for *LongNumbers* (Listing Eleven, page 80) provides procedures that input, manipulate, and output the *LONG* data type.

LongClear simply clears all elements of the array to 0. *LongAdd* (*LongSub*) is a multiple precision routine that uses an integer for a *carry borrow( )* flag. The idea is to index through the array, calculating the sum (difference) while checking for any overflow (underflow); such an overflow (underflow) causes the *carry borrow( )* flag to be set and the result to be adjusted. This *carry borrow( )* is then figured into the next digit's calculation.

The conversion routines *CardToHLLong* and *LongToCard* use the standard hexadecimal/decimal conversion algorithms, with the addition of range checking. *LongToCard* checks the range of the *LONG* and returns *FALSE* if any of the four most significant digits are anything but 0. *LongToInt* is a bit more complicated because there are two possible in-range conditions: either all unused bits must be 0s (positive integer), or all must be 1s (negative integer).

LongInc and *LongDec* use *LongAdd* and *LongSub* respectively, as well as *CardToLong*, to increment or decrement a *LONG* data type by any value in the *CARDINAL* range. *LongCompare* uses the standard comparison algorithm often used to compare strings.

LongPut and *LongWrite* cause output of an array of integers as hexadecimal numbers. They both use an internal filter routine called *GetDigit* to trap integers outside the hexadecimal range. The *Size* parameter is used to allow *LongPut* and *LongWrite* to output only a portion of the number in cases where a small hexadecimal number is stored as a *LONG* or to output extra long strings of hex digits for the S-records.

StringToLong converts an array of characters into a variable of type *LONG*. Error checking is done by the *IsHEX* routine, and *GetHEX* handles the digit-by-digit conversion. The two address-bounds routines use the set operator *IN* to force a *LONG* to specific address boundaries.

CmdLin2

I wrote the command line parser (Listing Twelve, page 80) as an experiment: I wanted to see just how flexible the Modula-2 pointer structure really was. My conclusion is that it is just as flexible and powerful as the pointer structure in C and much easier to understand.

This module could not have been written in Pascal because Pascal pointers can reference only vari-

ables dynamically created by the standard procedure *NEW*. Modula-2 pointers can be made to point to any data type.

This routine parses the command line buffer of the operating system, which is referenced by an absolute variable at 80H, into an array of pointers to strings. (Absolute variables are another new feature of Modula-2 and allow any variable to be placed at a specific location in memory.) The parsing is done without even recopying the buffer by setting a pointer to the beginning of each argument and a null terminator at the end (replacing the space that normally separates command line arguments). After all arguments have been so processed, *ArgV* is set to point to the pointers. (See Part 1 of this series of articles for a description of how the pointer is used to retrieve the arguments—C programmers will feel right at home.)

The CmdLin2 implementation uses a looping construct that is new to Modula-2: *LOOP . . . END*. This construct has two useful variations: the first is an infinite loop; the other, by using the optional *EXIT* statement, allows termination anywhere in the loop—even allowing multiple terminations. I know of one university instructor who asks his students to prove that *WHILE* or *REPEAT* are inappropriate before they are given leave to use the *LOOP*. Some authors refer to it as an unstructured loop. I tend to agree, instead, with Donald Knuth, who feels that all constructs—even the lowly *GOTO*—have an inherent structure; if that structure matches the structure of the problem, that's the one to use.

In CmdLin2, there are three *EXIT* statements in the *LOOP . . . END* statement. I went through many trials using *WHILE* and *REPEAT*, extra Boolean variables, and all the usual so called structured "tricks," but none were as clear and simple as the *LOOP* (yet still reliable under all conditions of input).

While reading through a Modula-2 textbook recently, I came across several examples of a small program fragment that was supposed to read integers and add them to a sum and then stop when something other than an integer was read. All three of the examples given either used two read statements, tested the same condition twice, or both. These were supposed to be examples of the correct way to use *WHILE* and *REPEAT* loops but were a perfect example of trying to shoehorn the problem to fit the structure. Such examples appear repeatedly in programming texts.

Modula-2's *LOOP . . . EXIT . . . END* construct provides a simple and elegant solution:

```
(* ReadInt and Done are imported    *)
(* from the standard module InOut   *)
(* Done is set TRUE if ReadInt      *)
(* is successful.                   *)
sum := 0;
LOOP
   ReadInt (num);
   IF Done THEN
      sum := sum + num;
   ELSE
      EXIT;
   END;
END;
```

Parser

The name for this module is really a bit of a misnomer because all it does is split up the 68000 source code into its components: *LABEL, OP-CODE,*

(continued from page 45)

SOURCE-OPERAND, and *DESTINATION-OPERAND* (Listing Thirteen, page 80). The algorithm is quite primitive in that it merely scans the line from left to right looking for the various parts and transfers them into variables called *Label*, *OpCode*, *SrcOp*, and *DestOp*. These variables are arrays of characters defined in the definition part of this module. The location of each item is noted for later use in the error handling module so that the exact location of any error can be pointed out.

The most convoluted part of the scanning process is picking out delimiters, especially when the normal delimiter characters get imbedded within parentheses or quotes. The problem is handled by a couple of flags. *ParCnt* keeps track of (possibly nested) parentheses counts, and *InQuotes* becomes *TRUE* inside quotes; both are used to prevent incorrect detection of delimiters. Parser will flag an error if any identifier or expression is too long. Labels and opcodes are limited to eight characters, and operands (including expressions) are limited to 20 characters.

SymbolTable

The implementation of this module (Listing Fourteen, page 84) hides a data type called *SYMBOL* and variables called *SymTab*, *Top*, and *Next*. This is an excellent example of the way Modula-2 allows you to separate lifetime and scope. These variables must exist during the entire time that the program is running, as they pass on information gathered in assembly pass 1 to assembly pass 2. Yet, at the same time, you don't want any access to these variables except through the symbol table routines. To allow Pascal variables to exist for the life of the program, they would have to be declared as global, at the risk of side effects from unplanned access. Modules provide a visibility wall between their contents and the rest of the program. *SymTab*, *Next*, and *Top* cannot be accessed from outside SymbolTable (limited scope), but they exist for the life of the program (global lifetime).

The *FillSymTab* routine simply adds a *SYMBOL* (a record consisting of an identifier and a *LONG* number) to the next open position in *SymTab* and returns an error if no room exists. *SortSymTab* uses a Shell sort to place the identifiers in alphabetical order for easy access. Notice in the *Swap* routine that entire records can be assigned in one statement in Modula-2. Not only is this more convenient than assignment one element at a time, but it is also more efficient: The compiler is able to use fast and compact assembly-language routines to copy the data into the new variable.

ReadSymTab uses a binary search to find quickly the value associated with any identifier. If the symbol is not found, *ReadSymTab* returns *FALSE* to the calling program. It is here, also, that duplicate symbol table entries are flagged (to do it in *FillSymTab* would require sorting and searching the table after each entry—which is hardly worth the extra time!).

ListSymTab really only returns one entry in the table and is used by the Listing module to provide a sym-

bolic reference table of identifiers at the end of the program.

OperationCodes

X68000 is a "sort of" table-driven assembler (I'm gonna get lots of objection to this one!). The mnemonics and data that are used to derive the opcode bit patterns and all addressing mode information come from a file called OPCODE.DAT. This file must exist on the default disk any time X68000 is run because the data must be read into the lookup table used in the binary search routine to find the instructions. X68000 is not a true table-driven assembler because it lacks the flexibility to accept tables of various processors and the table (OPCODE.DAT) is not a text file.

In Modula-2, implementation modules may optionally specify an initialization part. This initialization is run on program start-up, before the main program runs. The purpose is to set initial conditions within the module. The initialization for OperationCodes (Listing Fifteen, page 86) opens the file OPCODE.DAT, and reads the data into an array called *Table68K*. This data file is stored in compact binary format. *Instructions* is the only routine in OperationCodes; it uses a binary search routine to find the correct mnemonic opcode in *Table68K*. If found, its bit pattern, as well as two *SET*s consisting of (enumerated) addressing modes, are returned to the calling program. If the opcode mnemonic is illegal (that is, not found), an error is flagged by the error handling routine in ErrorX68.

InitOperationCodes

This program module (Listing Sixteen, page 86) is not actually part of X68000 but merely creates the data file OPCODE.DAT described above. It contains most of the same declarations and definitions that OperationCodes contains as their data types and variable have to match exactly.

The lookup table for the mnemonics is created by this program one mnemonic, bit pattern, and addressing mode at a time. There are 118 mnemonics (*ABCD* to *UNLK*), and each is assigned one element of an array. Each element of the array is a four-field record. After data is assigned to the array properly, it is written to a disk file using the *WriteRec* procedure.

Note: The *WriteRec* procedure may not be available on all implementations of Modula-2 but may be added easily by the programmer. It makes use of the generic parameter type *WORD*. Two possible implementations are:

For machines, such as the PDP-11 and many microcomputers, in which *TSIZE (WORD) = 2:*

```
WriteRec (f : FILE; Rec : ARRAY OF
                                WORD);
  VAR
    i : CARDINAL;
    ptr : POINTER TO CHAR;
  BEGIN
    ptr := ADR (Rec);
    FOR i := 0 TO HIGH (Rec) DO
      Write (f, ptr^);
      INC (ptr);      (* move
                  pointer to next byte *)
    END;
  END WriteRec;
```

For machines where *TSIZE (WORD) = 1* and a WriteWord procedure exists:

```
WriteRec (f : FILE; Rec : ARRAY OF
                                WORD);
  VAR
    i : CARDINAL;
  BEGIN
    FOR i := 0 TO HIGH (Rec) DO
      WriteWord (f, Rec);
    END;
  END WriteRec;
```

Similar routines can be developed for reading records, as required by the OperationCodes initialization.

Some libraries provide procedures to read or write multiple bytes to a file. These usually require the location (an address) and size (in bytes) of the record. In this case, Modula-2's low-level facilities may be used to read or write the record:

```
WriteNBytes (f, ADR (rec), SIZE (rec));
```

CodeGenerator

This module (Listing Seventeen, next month) does more than is suggested by its name. The definition module of CodeGenerator lists three procedures: *BuildSymTable*, which "generates the code" for pass 1 (that is, feeds the symbol table); *AdvAddrCnt*,

68K ASSEMBLER
(continued from page 47)

which increments the address counter after each instruction is analyzed (used in both passes); and *GetObjectCode*, which (with the help of many other procedures in CodeGenerator and SyntaxAnalyzer) figures out the machine code and returns it to the main program as three *LONG*s. I will (as far as possible) try to present the procedures in the order that the code would pass through them during assembly. This will mean some references to SyntaxAnalyzer, as these modules are tightly coupled and are used in both passes.

BuildSymTable is used only during pass 1, and, as was mentioned above, it is involved with creating the symbol table. It does nothing (returns immediately) if there is no opcode. The cascaded *IF ELSIF ELSE* statement determines, first, if any assembler directive is present, and if so, specifies the amount of memory reserved (that will vary—EQU reserves no memory, whereas *DS* may reserve any amount). Next, if there is any la-

Modula-2 Porting Experience

Background

Modula-2 is a relatively young language, and efforts to standardize both the basic language and the library are still underway. Modula-2 was developed at the Swiss Federal Technical Institute (ETH) in the late 70s by a team headed by Niklaus Wirth, the creator of Pascal.

The original language definition was contained in *Programming in Modula-2* by Niklaus Wirth, 2d ed. (New York: Springer-Verlag, 1983). On March 3, 1984, Wirth released a brief entitled "Revisions and Amendments to Modula-2," which outlined several clarifications, revisions, and extensions to the language itself. This brief was published more recently in the January/February 1985 issue of *The Journal of Pascal, Ada, & Modula-2;* the changes have also been incorporated into the third edition of Wirth's book. Most microcomputer compilers have not yet implemented these changes or have implemented only some of them.

The library, too, exists in several versions. In his original work, Wirth postulated a standard minimum library—much of it for the DEC PDP-11 minicomputer. Since then, a much more extensive library has been developed for the Lilith, the Modula-2 bit-slice microcomputer developed at ETH. One of the earliest (popular) microcomputer implementations of Modula-2 was by Volition Systems, which produced compilers for several computers including the Apple II, IBM PC, and Sage. Several textbooks have been based on Volition Systems' compiler and library, including those by Gleaves, Wiener and Sincovec, Wiener and Ford, and Knepley and Platt. Finally, the Modula User's Society (MODUS) has been working on what it hopes will be a universal library.

Porting Among Systems

By and large, the compilers themselves are pretty much compatible (despite the amendments mentioned above). Virtually all microcomputer implementations use 16 bits for *CARDINAL*, *INTEGER*, and *BITSET* data types; most implement strings as *NULL* terminated *ARRAY OF CHAR*. (Interface Technologies' is the only product I know of that implements strings Pascal-style—presenting a serious impediment to portability.) All the standard data types, structures, loop constructs, and operators are invariably provided. Some compilers omit or alter the low-level facilities for concurrent processing, but most programmers will have limited use for these advanced features.

Differences in the library, particularly in file handling, are where most of the problems lie. The four systems I am familiar with (those from Hochstrasser, Volition Systems, Interface Technologies, and LogiTech) have three significantly different approaches to files. Hochstrasser's and Volition Systems' are essentially similar and follow the earlier (PDP-11) work at ETH; LogiTech's follows that for the Lilith quite closely; whereas Interface Technologies' seems to go its own way. On the plus side, all implementations provide the high-level I/O modules suggested by Wirth (Terminal and InOut) as well as the the floating-point library (MathLib0). Most provide essentially similar conversion libraries for changing strings to numbers and vice versa.

Porting X68000 from Hochstrasser (CP/M) to LogiTech (MS DOS)

Hochstrasser supplies several file-handling modules with its compiler. I chose to use the one called Files because I knew it to be nearly identical to a module of the same name supplied by Volition Systems. The Files module is flexible and powerful in that it allows random or sequential access; it also provides methods to read or write complete data structures. LogiTech supplies one file-handling module—FileSystem, which is capable of all the same operations as is Files, along with text-oriented streams (supplied as a separate module called Texts in the Hochstrasser and Volition Systems libraries).

The first (and easiest) step in porting to a new compiler is to change the names of the various procedures to match the new library. In this instance, for example, *Read* became *ReadChar*, whereas *WriteRec (f, RecordVariable)* had to be converted to *WriteNBytes (f, ADR (RecordVariable), TSIZE (RecordVariable), Written)*. All these kinds of changes are obvious from reading through the documentation of the compiler systems, and they present few problems.

Several of the necessary changes are not so obvious. Both systems supply a *Create* procedure, which (as the name suggests) is for creating a new file. LogiTech's system, however, creates only a temporary file, which disappears as soon as the file is closed. To open an existing file, Hochstrasser supplies the procedure *Open* and LogiTech supplies *Lookup*. *Lookup* is a general-purpose procedure that, depending on a *BOOLEAN* parameter, will either open an existing file or create a new one. Because the assembler must often create a new version of an existing object file, I found it necessary to call *Delete* before using *Look-*

bel present (whether there was an assembler directive or not), an entry is made in the symbol table. That entry will consist of the current value of the address counter except in the case of the *EQU* assembler directive.

Notice that an error due to a full symbol table will be detected here—this is a fatal error and will cause the assembler to abort. You must then split up your program into two or more modules. If there was no assembler directive (pseudo-op), *GetOperand* and *GetInstModeSize* (procedures from the SyntaxAnalyzer module) help to determine the size of the operands. The special *QUICK* mode instructions must be taken into account before determining how far the address is going to have to be advanced by this instruction.

During pass 2, *GetObjectCode* first checks if there is any opcode, and it returns without doing anything if there is none. After making note of the size extension of the opcode, control is passed to *ObjDir*, which handles code generation for assembler directives (see detailed description later). Phase errors are checked by com-

up to create a new file.

The Hochstrasser Files module implements the type *FILE* as an opaque (hidden) type (that is, as a pointer to some structure defined in the implementation module). The opaque type *FILE* allows the programmer to use files with no knowledge of their underlying structure. LogiTech's system, taking the same approach as that for the Lilith, implements type *FILE* as a record that is specified in the definition module. This forces the programmer to know the intimate details of the file structure and also has one very (for me at least) unexpected and disturbing consequence: The type *FILE* must always be passed as a VAR parameter. If it is passed as an ordinary value parameter (as can be done with the opaque pointer type), the calling program is unable to get information back to the FileSystem module, resulting in rather odd run-time errors. It took quite a bit of head scratching to puzzle that one out. With all the changes in place, the file systems on the two versions of X68000 produce outwardly identical results.

LogiTech's compiler passes all command-line parameters to the first *ReadString* call in the main program, which obviates the need for my *CmdLin2* procedure. From the user's perspective, the effect created is essentially similar: The assembly file name can be entered either as the program is invoked or in response to a prompt once the program has started.

The number conversion routines supplied by the two compilers differ both in name and in behavior. In one case (*CARDINAL* to *STRING*), I had to add an extra parameter (specifying length), whereas in another (*STRING* to *CARDINAL*), the behavior of the equivalent LogiTech procedure was sufficiently different that I had to write a new conversion routine.

Other Factors Not Related to Portability

At the college where I work, one of my colleagues has dubbed the IBM PC "Miss Piggy" for its propensity to gobble up memory. It seems that with every application you care to name, the PC requires much more memory than does any other computer. Compilers are no different in this regard.

The original work on X68000 was done on a standard CP/M system with 64K of memory and two 1.2-megabyte floppy-disk drives. When I started work on the port to the MS DOS environment, I tried using a standard IBM PC with 256K of memory and two 360K floppy-disk drives. I soon found that the standard PC was not up to the task.

Upon moving to a machine equipped with a 20-megabyte hard-disk drive, I was able to start compiling, and all but two of the modules would compile. On the InitOperationCodes module, the compiler aborted with an "out of memory" error. This was the program module that initialized the mnemonic lookup table, and then wrote it to a file—it was impractical to split it into several modules. The only solution was to expand the memory to 512K. Even with the extra memory, however, an implementation restriction still prevented several of the procedures from compiling because they were too long. Remember, this is the same code that compiled without a whimper on a 64K CP/M system. Eventually I had to split the InitOperationCodes main program into four procedures and also split the *MergeModes* procedure (from the CodeGenerator module) into four. Despite the length of these procedures, splitting them made no sense from the point of view of program logic.

The Hochstrasser compiler does no error checking at run time, whereas LogiTech's checks for integer/cardinal overflow, array range errors, CASE out of range, and so on. It seems that all the type transfers used in X68000 cause some range errors because, before I could get the program to work, all the error checking had to be turned off. This is easily done with the LogiTech compiler (via compile-time option switches), and it causes no problem whatever to the accuracy or internal error checking done by X68000.

Just out of curiosity, I timed the compilers on several large modules. Both compilers use four passes, which are overlayed from the disk. The Hochstrasser linker produces an executable file directly, whereas LogiTech's requires an additional step for this. Even with the hard disk and 512K installed on the IBM PC, the LogiTech compiler was only about 20 percent faster than was the Hochstrasser compiler.

The Verdict

Porting X68000 to the IBM PC using the LogiTech compiler was a relatively painless and quick process. The total time expended was roughly 15 hours—much of which was spent getting used to a new compiler. If you are already familiar with your compiler, you should have little trouble making the conversion in a single afternoon.

True to form, the compiled code is 47 percent larger on "Miss Piggy" than it is on the CP/M system. Execution speed (of X68000) is essentially the same on the CP/M and MS DOS versions.

(continued from page 49)

paring the pass 1 address count (from the symbol table) with the pass 2 address count for any line that has a label. The two instruction types that use relative addressing modes are handled as a special case because of the odd requirements placed on them. (There is no automatic selection of the most efficient branch length. The assembler assumes the worst case and produces long branches unless explicitly told to use the short form. Full range checking is done for either long or short branches, however.) Object code for the balance of the instructions is produced in *MergeModes* (described next).

Although the 68000 instruction set is very orthogonal (regular), there are a few instructions that have to be handled as special cases. That is the purpose of the *CONST* definitions at the top of the module: they are *BITSET* constants representing several operation codes. These are used within the *MergeModes* procedure to take care of the special cases before the more common addressing modes are handled.

MergeModes is not exported from the definition module of CodeGenerator but is used by the *GetObjectCode* procedure to combine information from several sources to produce the hexadecimal machine code. *MergeModes* is at the heart of the code generation process. Many 68000 instructions use a format that is some variation on *ADD Dn,<ea>*. The effective address <ea> may be any of 12 addressing modes, although few instructions use all 12 modes. There are four basic groupings of these modes—*Data*, *Memory*, *Control*, and *Alterable*—which may be combined in various ways. The local procedures *EffAdr* and *OperExt* determine the bit patterns needed for the effective addressing mode being used along with any operand extension needed for that addressing mode. (For example, *MOVE D3,600(A2)* requires that bit patterns of 000011 and 101010 be inserted into the opcode for the source and destination operands, and an extension word of 0000001001011000 needs to be tagged on after the opcode for the 600 offset.)

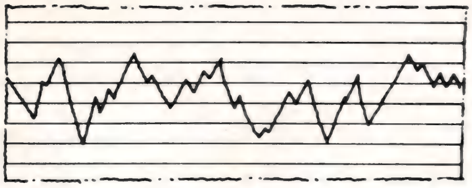
The bulk of *MergeModes* cross-checks the addressing mode found by *GetOperand* with the modes that are allowed for the current instruction (information from OPCODE.DAT and the OperationCodes module). Any errors are passed onto the ErrorX68 module, where verbal error messages are displayed on the console. If no errors are detected, various bits and pieces of information are combined to produce the machine code for the instruction.

An example will illustrate typical operation for the complete sequence needed for code generation. This comprises two steps: first converting the source code to an intermediate language consisting of sets, enumerations, and various integral values; and then combining the elements of this intermediate language into Motorola 68000 machine language.

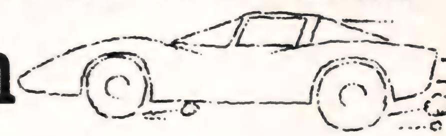
```
ADD (A2),D6  ;Add the data word
                  addressed by A2
             ;to data register D6
```

When the Parser module is finished with this instruction, you are left with three character strings: *ADD*,

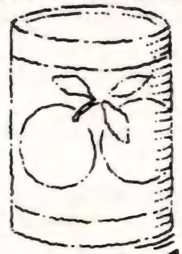
VEDIT® Plus Text Editor



The Navy charts new concepts with it... GM engineers the future with



it...



National Can preserves facts with it... GE has bright ideas with it... Here's why you shouldn't be without it.

Every day, VEDIT PLUS helps thousands of programmers, writers and engineers get down to business.

So why do people who could have ANY text editor prefer ours? For a lot of reasons, including:

- CAPACITY—With VEDIT PLUS, file size is never a problem. And virtual disk buffering simplifies editing of even the largest files.
- FLEXIBILITY—VEDIT PLUS lets you edit up to 37 files simultaneously. So you can cut and paste. Edit programs. Edit text. Even perform numerous search/replace functions in several files without user intervention.*
- CUSTOMIZATION—With VEDIT PLUS, you can create your own on-line editing functions with keystroke macros. Develop your own on-line help screens. Determine and revise your own keyboard layout easily.
- SPEED—VEDIT PLUS not only works hard, it works fast. Faster, in fact, than any other text editor on the market.
- EXPERIENCE—Six years ago, CompuView revolutionized the concept of microcomputer text editing. And we've been improving our products and services ever since.

Special Offer: Order a VEDIT PLUS text editor for $225 and we'll include our V-PRINT™ document formatter—a $120 value—absolutely free.

Call CompuView today at 313/996-1299. You'll be in good company.

CompuView®

CompuView® Products Inc., 1955 Pauline Boulevard—Suite 300, Ann Arbor, Michigan 48103, TELEX 701821

Available for PC DOS, MS-DOS, CP/M, CP/M-86.

*Free sort, compare, print and main menu macros included; optional 8080-8086 translator or mailmerge, $50 each.

Circle **no. 122** on reader service card.

(continued from page 50)

(A2), and *D6*. Parser passes *ADD* to OperationCodes, which returns three sets: *{15, 14, 12} —> 1101 0000 0000 0000*, *ModeA{OpM68D}*, and *ModeB{EA05y}*. The first set is the raw bit pattern for this instruction—that is, the bit pattern without operand-size bits and operand addressing-mode bits added. The other two sets specify the addressing modes that the *ADD* instruction can use and are designations internal and unique to X68000.

The Parser passes (A2) and D6 to the *GetOperand* procedure in SyntaxAnalyzer (through CodeGenerator), where they are analyzed. A record of type *OpConfig*, which contains the mode, value, size, and various other information, is provided for each operand. In this example, those records would contain the following data:

```
(A2)  Mode —> ARInd
                    ;register indirect
      Value —> none
      Loc —> location on source line
      Rn —> 2
      Xn —> none
      Xsize —> none
      Xtype —> none
D6    Mode —> DReg   ;data register
      Value —> none
      Loc —> location on source line
      Rn —> 6
      Xn —> none
      Xsize —> none
      Xtype —> none
```

MergeModes will take all pertinent information from the above to produce the machine code or, if there are inconsistencies, produce error messages.

An *IF* statement checks for each of the possible addressing modes. For example, the *IF OpM68D IN AddrModeA* statement would be used for the *ADD (A2),D6* instruction because *ADD* resulted in *ModeA{OpM68D}* being returned from OperationCodes. The *Dest.Mode* is checked to see that it is *DReg* (it is), so the *Dest.Rn (6)* is shifted left by 9 and *OR*ed with *Op*. Because shift left is not provided in Modula-2, I used multiplication to accomplish the same thing (multiplication by 2 is the same as shift left 1). Because of a principle called strength reduction, this multiplication (by a power of 2) is not nearly as inefficient as you might think. As part of this same *IF* statement, the size bits are *OR*ed into place depending on the size suffix placed on the instruction.

Because there is no size suffix on *ADD (A2),D6*, size *WORD* is assumed. The *IF EA05y IN AddrModeB* will fill out the operation with more error checking and a call to the *EffAdr* local procedure mentioned above. This procedure checks that the mode used is consistent with the instruction, then uses bitwise *AND/OR* to append the correct bits. *OperExt* would have nothing to do on this instruction because neither *ARInd* or *DReg* require an operand extension.

All instructions follow a similar format: source line —> line parts (label, opcode, operands) —> intermediate language (bitsets, enumerations, and so on) —> machine code. Any or all of the processes involved in reaching these states can result in error messages if the source line does not conform to correct 68000 syntax.

The hidden procedure *ObjDir* is the assembler directive equivalent to *MergeModes*—it produces the code for the directives. It is essentially similar to the cascaded *IF ELSIF ELSE* statement that handles pseudo-ops in the *BuildSymTable* routine except that it has to generate the code, which means determining values and setting object code lengths. This procedure also handles ASCII strings. I'm not at all satisfied with this section of code, and if you think it looks like an afterthought, you're right! The awkwardness results from having to pass the string (which is converted to *LONG*) to the output modules as a 2-byte opcode and two 4-byte operands. The other alternative was a major rewrite, which will have to wait until Version 2 (when I will have to rewrite some of the code to accommodate a linker, anyway).

SyntaxAnalyzer

The procedures within SyntaxAnalyzer (Listing Eighteen, next month) are not visible to any other module except CodeGenerator (it was originally part of the same module). Its purpose is mainly to analyze the operands of the instructions and to determine their value and their mode.

CalcValue and *GetValue* work together, as their names suggest, to determine the value of any operands that have a value. These include decimal numbers (0−65535); hexadecimal numbers (0−$FFFFFFFF); single quoted ASCII literals; the symbol for the current value of the program counter (*); and identifiers, which may represent any value. *GetValue* contains a simple left-to-right expression evaluation loop that recognizes only addition and subtraction operations. It hardly has the elegance of a recursive descent expression parser, but it is simple and compact. *GetValue* uses the *LOOP . . . END* construct with three conditional *EXIT* statements. Although this could have been done with a *REPEAT* loop, the termination condition would have been awkward, with six terms, three *AND*s, and three *OR*s. I rest my case!

Two procedures, called *GetSize* and *GetAbsSize*, determine the size of operands (*BYTE*, *WORD*, or *LONG*) by looking for a suffix of *.B*, *.W*, or *.L*. If no suffix is present, size *WORD* is assumed, as required by the Motorola syntax. The *GetAbsSize* procedure actually creates a slightly nonstandard syntax for this assembler. Most 68000 assemblers will automatically choose the *WORD* absolute addressing mode for addresses in the ranges of 8000−0 and 0−$7FFF and *LONG* absolute addressing for higher addresses. X68000 will always use full 32-bit addressing unless specifically instructed to do otherwise. The nonstandard syntax is *MOVE D0,$6000.W*.

The *GetInstModeSize* uses a *CASE* statement to return the size of the object code, both in terms of address count and in terms of number of hexadecimal digits required.

GetOperand is the workhorse of SyntaxAnalyzer, as it is responsible for performing lexical analysis on the rather complex and varied operands used in 68000 assembly language. Not much elegance here—this routine simply looks for all the possible addressing modes. When it finds a good one, it returns all necessary information in the form of the record described above under CodeGenerator. If *GetOperand* finds an impossible addressing mode or an out of range register number, it uses ErrorX68 to report the error to the user console.

GetMultReg is a routine that sorts out the *MOVEM* instructions. This instruction is like a multiple *PUSH* or multiple *PULL* operation. *MOVEM D0-D7/A0-A6,-(SP)*, for example, will push all 68000 address and data registers onto the stack with one instruction. This is accomplished by a mask that follows the actual instruction, where each bit in the mask represents a register; if the bit is set, that register gets moved. The job of *GetMultReg* is to produce that mask from the register list. The – indicates a range: D0–D7 means all registers between D0 and D7 inclusive, and the / is just a separator. My strategy was to use a flag and an enumeration type along with nested *IF*s to keep track of what state the calculations were in. That made it easy to detect errors because it would cause a transition to an illegal state. Just to complicate matters, the mask has to be inverted in certain cases. That function is taken care of by a conditional subtraction as the mask is constructed.

Listing

The Listing module (Listing Nineteen, next month) creates the formatted program listing with object code and source code together in the usual format. The *StartListing* procedure does nothing but print a heading and initialize the page count and line count variables. *WriteListLine* does what its name implies—writes one line of listing to the file. Unused object code fields will be skipped automatically (no address is entered in the case of an *EQU* pseudo-op, for example). The *LongPut* routine from LongNumbers is used to write all object code (in hexadecimal) to the file.

Modula-2 file libraries do not usually contain any way to write a string to a file (*InOut* and *Texts* allow this, but that's another story), so I had to write a small procedure to do that. The *WriteStrF* outputs any string to a file and is used throughout Listing.

The *WriteSymTab* procedure uses information imported from the SymbolTable module via *ListSymTab* to output a symbolic reference table at the end of the listing. *ListSymTab* will return the nth entry in the symbol table each time it is called. Both *WriteListLine* and *WriteSymTab* make use of a procedure called *CheckPage* to form-feed to the next

(continued from page 53)

page and print a new page number. The listing file created by this module may be dumped to a standard printer with the *PIP* command.

Srecord

Creation of S-record files (Listing Twenty, next month) is a bit more complicated than creation of listing files. Count and checksum bytes must be calculated, and it is usual to split the source code into equal-length records unrelated to the length of the 68000 instructions (that is, some instructions may be split with half on one line of the file and the rest on the next line).

My strategy in this was to accumulate source code until there was enough to output a 16-byte record, output that record (saving any extra bytes accumulated for the next record), then go back to accumulating more. This necessitated two storage arrays and two indexes for accessing them: *Sdata/Sindex* and *Xdata/Xindex*. Another complication is that records should start on boundaries divisible by 16 whenever possible. Only three procedures are exported from the definition module of Srecord: *StartSrec*, *WriteSrecLine*, and *EndSrec*. Several other procedures that are hidden within the implementation module do much of the work.

StartSrec creates the *S0* (header) record. This record consists mainly of the source file name as ASCII characters represented in hexadecimal format. However, all S-records must have an address (this is always 0 for the header record), a byte count, and a checksum. The byte count includes a count for the address and the checksum; the checksum is the complement of the 1-byte residual of the sum of the address, count, and data.

The *WriteSrecLine* procedure returns immediately if there is no address to write out (this occurs on blank source lines or for *EQU* statements only). Next, *Xdata* is transferred to *Sdata* if there was anything left over from the previous call to *WriteSrecLine*. If for any reason the address count passed into *WriteSrecLine* is different from the internal count, any existing data is output as a complete S-record, and a new record is started. This would occur any time a new *ORG* statement is encountered in the source code. Finally, each of the *ObjOp*, *ObjSrc*, and *ObjDest* (object code for opcode, source, and destination, respectively) are appended to *Sdata*. If *Sdata* now contains more than 16 bytes, the record is written out to the file (this is detected by *AppendData*, returning *FALSE*). Any excess object bytes are retained in *Xdata*.

The information in *Sdata* and *Xdata* is retained between calls to *WriteSrecLine* because these variables are declared within the module (not within a procedure). These variables cannot be seen outside the Srecord module (local visibility) but remain in existence throughout the lifetime of the program (global existence). In Pascal, lifetime and visibility are tied together: If a variable exists, it is visible; if it is not visible, it no longer exists. That prevents local variables in Pascal from retaining values between calls—a very useful feature, as illustrated here.

The *EndSrec* procedure outputs any data left over from the final call

to *WriteSrecLine* and then outputs a fixed *S8* trailer record.

Actually, this even-boundary and consistent-length business is not required by the S-record format because each S-record is totally autonomous in that it begins with both its own starting address and a byte count. It is traditional for Motorola S-records to be formatted as described above, however.

ErrorX68

Most error handling (except that involving files) is done by the ErrorX68 module (Listing Twenty-One, next month). This module defines an enumeration type that provides 12 named error types. The procedure *Error* outputs the line count (source line where the error was found), the source line itself, an arrow pointing to the error, and a verbal error message. The program is then suspended until the operator acknowledges the error by pressing any key on the console keyboard. If greater than 500 errors occur, the program is terminated with an appropriate message. After pass 2 is completed, WriteErrorCount outputs an END OF ASSEMBLY message to both the console and the listing file.

Compiling X68000

Because many of the modules within X68000 interact in complicated ways, the order of compilation is critical. Specifically, if ModuleA imports objects defined in ModuleB, it is clear that ModuleB must be compiled first. In all cases, a definition module must be compiled before its implementation module can be compiled; how-

Compilation Order for X68000

1. CmdLin2.DEF, CmdLin2.MOD
2. LongNumbers.DEF, LongNumbers.MOD
3. Parser.DEF
4. CodeGenerator.DEF
5. SyntaxAnalyzer.DEF
6. SymbolTable.DEF, SymbolTable.MOD
7. OperationCodes.DEF, OperationCodes.MOD
8. Listing.DEF, Listing.MOD
9. Srecord.DEF, Srecord.MOD
10. ErrorX68.DEF, ErrorX68.MOD
11. Parser.MOD
12. SyntaxAnalyzer.MOD
13. CodeGenerator.MOD
14. X68000.MOD

Table 1

(continued from page 55)

ever, it goes further than that. The definition module of Parser, for example, defines two types (*TOKEN* and *OPERAND*) that are used in SymbolTable, CodeGenerator, SyntaxAnalyzer, and others. Therefore, Parser.DEF must be compiled before any module that depends on it. If the correct order is not followed, the compiler will produce "undefined identifier" errors. Many similar situations exist in any nontrivial Modula-2 program.

The compilation order shown in Table 1, page 55, avoids any problems but is not the only possible ordering arrangement. (A harmless circular reference exists between Parser and ErrorX68 because each imports objects from the other. This only affects the order of execution of their respective initialization parts and causes no problems of any sort.)

Modula-2 Design Strategy

The implementation module often hides details not apparent in the definition module. Obviously, the algorithm is encapsulated inside the implementation module, but it goes further than that. Constants, types, variables, and procedures that are not visible from the definition module may contribute an important part to the module's function. (For example, the *GetDigit*, *IsHEX*, and *GetHEX* routines from LongNumbers as well as the *LineParts* procedure from Parser are unknown to the definition modules, and hence they need not be known by any programmer using these modules.)

Additionally, the module initialization may play an important role in the structure of the program, as in the SymbolTable module when *SymTab* is cleared and the indexes for access to the table are set to their starting points or as in OperationCodes where data is brought in from a file. Finally, modules may be used to control visibility and lifetime fully: Nothing is visible outside a module unless it is exported, but variables belonging to library modules exist throughout the life of the program. These features allow large programming projects to be constructed of modules of only a few closely related procedures. Debugging is easier, and maintenance is finally possible!

Conclusion

Although X68000 is a fully functional program, I do not consider it a completed project as several areas could use improvement. Expression and string evaluation should both be improved. The first steps would be to expand and improve LongNumbers to include multiplication and division and to improve efficiency. This can wait until Modula-2 compilers conform to the new standard and add long integer and cardinal data types. Constant strings should be expanded to at least 80 characters per line. That is harder than it sounds because of the way parameters are passed to the Listing and Srecord modules.

Finally, there is the matter of a linker. To adapt the assembler to provide relocation information will require some rewriting of existing code and some new code; however, many of the modules could be reused almost intact. Then there is the Linker itself—not a trivial task either. If any readers have further suggestions on how X68000 can be improved, please pass them along to me. Better yet, make the changes yourself and hand the program back into the public domain in improved form.

Availability

The following is available directly from the author for $20 (U.S.):

1. A 25-page X68000 *User's Manual* that includes operating instructions for the program as well as a description and example of a method to use the assembler to link several modules.
2. An 8-inch CP/M SSSD disk or a 5¼-inch IBM PC disk with executable program as well as complete source code and several 68000 assembly-language examples.

DDJ

(Listings begin on page 80.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

The Cryptographer's Toolbox

by Fred A. Scacchitti

The strongest cypher schemes use a combination of transposition and substitution.

The "Infinite Key Encryption System" article in the August 1984 issue of *DDJ* contains an excellent tutorial on encryption systems. At the time it was published, I read it with only passing interest, but about six months ago, I developed a need for this type of utility and so begins my story. The first thing I did was write a shell cypher program (cypher.c—Listing One, page 94). Because I had already written a generic file copy utility that allowed modifications during transfer, it was a simple matter to modify the argument passing to include multiple keys and add a *cypher( )* function call to encrypt the file with a simple exclusive-ORing algorithm (cypher1.c—Listing Two, page 94). Although this method did encrypt the file and allowed for easy decryption (using the same run string), there were definite detectable patterns in the resultant file. These patterns, a function of the key period, were easily found in areas of repetitive characters (for example, a string of asterisks or spaces). Another drawback to this scheme was the inability to pass nonprintable characters in the run string, thereby limiting the number of encryption tokens. So, it was back to the drawing board.

Rereading the aforementioned article with renewed interest, I gained an insight into the methods and schemes of practical modern cyphers. I don't intend to cover these concepts, so if you're interested avail yourself of that article and those in its bibliography.

Although the article's tutorial portion was excellent, I disagreed with a few points on implementation. First, there was the code itself and the intimation that assembly language was required for reasonable speed. The MAC Assembler is used only with CP/M-80, and I wanted more portable source, so I decided to write it in C. Second, I felt that a random key of some prime length could be generated solely from the original key (cypher2.c—Listing Three, page 95). Although some keys may work better than others, a means to evaluate results can render this method functional. And last, I disagreed with the need for passing information in the encrypted file. It seemed unnecessary and cumbersome. My goal was to develop a method that worked entirely from the run string. Overall, though, I must commend the work done by John Thomas and Joan Thersites for presenting such a complete treatment of their topic.

The Ultimate Cypher

The ultimate cypher is like the ultimate weapon—no matter how sophisticated, an antiweapon (anticypher) can be developed eventually if there is a need. The user must make some judgments regarding needs and level of protection. The two algorithms mentioned above are relatively simple to implement and use. The same keys can encypher or decypher the file, and key order isn't important.

The experts (and it becomes quite obvious) point out that the strongest cypher schemes utilize a combination of transposition and substitution. When transposition is added however, the order of decyphering must be the exact reverse of encyphering. The last cypher module contains an algorithm for transposing the file tokens along with the random generated key encryption scheme (cypher3.c—Listing Four, page 95). This is a small price to pay for the added security.

The Need for Tools

As I progressed in my quest for the ultimate cypher/decypher algorithm, I became aware of the deficiencies of the standard CP/M utilities at my disposal, so I developed my own.

The first tool, fv.c (Listing Five, page 97), replaced my CP/M dump.com. Dump provides a continual onscreen display of the hex contents of a file. Because most encryption is performed on text files, it is beneficial to include the ASCII form along with the hex. And, because most algorithms use an exclusive-OR as the means of encryption, it is easy enough to dump two files and the exclusive-ORed difference between them.

©1985 by Fred A. Scacchitti, 25 Glenview Lane, Rochester, NY 14609.

The next tool, fstat.c (Listing Six, page 98) calculates and displays the statistical characteristics of the file. This tool scans the file, counting the occurrences of each element, and provides a 16 × 16 display of the distribution of characters. It calculates mean, median, mode, and range of the character distribution and displays its histogram. As you might suspect, each file type has a definite signature. In fact, after limited use of this utility, you will be able to recognize the histogram patterns for text, WordStar, and many other files.

While experimenting with various schemes, it became obvious that the most difficult file to disguise was one that contained a single byte for every entry or some sequential scheme. So, the next task was developing a utility, makef.c (Listing Seven, page 100), to generate a known sequential or unicharacter file of some user-defined length.

Finally, the last utility, sp.c (Listing Eight, page 100), was a search scheme I needed to look for repetitive patterns occurring within a file and to provide some information regarding location and depth of the repetition. It also includes the option to calculate the delta characteristics of a file to search for repetitive mathematical as well as character sequences.

Cypher.c—Listing One

The cypher shell program is provided for use "as is" or for user modification. It contains the argument-passing and file-handling source code needed to copy from an existing file to a new file via a 16K buffer, with a cypher function being called to encrypt the file. (If the file is less than 16K, the input file name may be the same as the output thus destroying the original contents.) I chose a 16K buffer because it should fit easily with most compilers. This value may be adjusted to meet individual compiler needs.

Any of the three algorithms that follow may be included or linked with this shell. Caution is recommended to ensure that the function name is appropriate for the method you use. The keys are passed in the CP/M command line and therefore are limited by its length as well as the argument-passing capability of the C compiler.

CRYPTOGRAPHER'S TOOLBOX

(continued from page 59)

Cypher1.c—Listing Two

This minimal cypher algorithm uses an exclusive-ORing scheme to encrypt the file with the keys passed. If the user employs keys of some prime length and performs multiple passes, the results can be quite difficult to decypher. Because the keys are limited to include only printable characters, you don't take full advantage of the 256 codes available for a byte.

Cypher2.c—Listing Three

Now something a little more difficult for the code breaker, an algorithm that grew out of the previous listing and generates a prime-length key for each user key passed. One of 50 prime values (between 1,009 and 1,999) is selected as a function of the key passed. The prime key is then generated using a simple summing-ANDing-exclusive-ORing algorithm, and the file is encrypted using this new key. If two or more keys are used, this method guarantees a cypher period in excess of 1,000,000, which is significantly larger than most text files.

The key-generation scheme is based entirely on the original key length and its contents, and I fail to see how this can be worked backward to regain the original, especially if multiple keys are used. Some keys will generate shorter periods within the prime length, but this is easily tested with the tools provided. I welcome feedback or suggestions for improving this algorithm.

Cypher3.c—Listing Four

Adding chaos to disorder has probably driven many a code cracker to drink, and this is just what I'm trying to accomplish. I have modified the cypher2.c algorithm slightly to test the first character of each key passed. If the key begins with a dash (-), then the buffer is transposed by some value between 2 and 17; otherwise, it encrypts the file using the algorithm as described above. This simple but effective method puts the

f-1	0000	2F	2A	09	63	79	70	68	65	72	31	2E	63	09	43	79	70	/*cypher1.c^Cyp
f-1	0010	68	65	72	20	6D	6F	64	75	6C	65	09	09	62	79	20	46	her module^^by F
f-1	0020	2E	41	2E	53	63	61	63	63	68	69	74	74	69	0D	0A	2A	.A.Scacchitti^^*
f-1	0030	2A	09	09	09	09	09	09	09	31	30	2F	31	30	2F	38	35	*^^^^^^^10/10/85
f-1	0040	0D	0A	2A	2A	0D	0A	2A	2A	09	53	69	6D	70	6C	65	20	^^**^^**^Simple
f-1	0050	63	79	70	68	65	72	20	6D	6F	64	75	6C	65	20	2D	20	cypher module -
f-1	0060	65	6E	63	6F	64	65	73	20	64	69	72	65	63	74	6C	79	encodes directly
f-1	0070	20	77	69	74	68	20	75	73	65	72	20	6B	65	79	73	0D	with user keys^
f-1	0080	0A	2A	2A	0D	0A	2A	2F	0D	0A	0D	0A	23	69	6E	63	6C	^**^^*/^^^^#incl
f-1	0090	75	64	65	20	3C	73	74	64	69	6F	2E	68	3E	0D	0A	0D	ude <stdio.h>^^^
f-1	00A0	0A	73	74	61	74	69	63	20	69	6E	74	20	69	2C	20	6E	^static int i, n
f-1	00B0	2C	20	6B	65	79	6C	65	6E	67	74	68	3B	0D	0A	0D	0A	, keylength;^^^^
f-1	00C0	63	79	70	68	65	72	31	28	62	75	66	66	65	72	2C	20	cypher1(buffer,
f-1	00D0	6E	75	6D	2C	20	63	6F	64	65	29	20	63	68	61	72	20	num, code) char
f-1	00E0	2A	62	75	66	66	65	72	2C	20	2A	63	6F	64	65	3B	20	*buffer, *code;
f-1	00F0	69	6E	74	20	6E	75	6D	3B	7B	0D	0A	0D	0A	2F	2A	0D	int num;{^^^^/*^
f-1	0100	0A	2A	2A	20	67	65	74	20	6B	65	79	6C	65	6E	67	74	^** get keylengt
f-1	0110	68	20	66	6F	72	20	65	61	63	68	20	6B	65	79	0D	0A	h for each key^^
f-1	0120	2A	2F	0D	0A	0D	0A	20	20	20	6B	65	79	6C	65	6E	67	*/^^^^ keyleng
f-1	0130	74	68	20	3D	20	30	3B	0D	0A	20	20	20	77	68	69	6C	th = 0;^^ whil
f-1	0140	65	28	63	6F	64	65	5B	6B	65	79	6C	65	6E	67	74	68	e(code[keylength
f-1	0150	2B	2B	5D	20	21	3D	20	4E	55	4C	4C	29	3B	0D	0A	20	++] != NULL);^^
f-1	0160	20	20	6B	65	79	6C	65	6E	67	74	68	2D	2D	3B	0D	0A	keylength--;^^
f-1	0170	0D	0A	2F	2A	0D	0A	2A	2A	20	65	6E	63	72	79	70	74	^^/*^^** encrypt
f-1	0180	20	74	68	65	20	66	69	6C	65	20	77	69	74	68	20	65	the file with e
f-1	0190	61	63	68	20	6B	65	79	0D	0A	2A	2F	0D	0A	0D	0A	20	ach key^^*/^^^^
f-1	01A0	20	20	70	72	69	6E	74	66	28	22	2D	65	6E	63	6F	64	printf("-encod
f-1	01B0	69	6E	67	2F	64	65	63	6F	64	69	6E	67	20	62	75	66	ing/decoding buf
f-1	01C0	66	65	72	5C	6E	22	29	3B	0D	0A	0D	0A	20	20	20	66	fer\n");^^^^ f
f-1	01D0	6F	72	28	69	3D	30	3B	20	69	3C	3D	6E	75	6D	3B	20	or(i=0; i<=num;
f-1	01E0	69	2B	2B	29	0D	0A	20	20	20	20	20	20	62	75	66	66	i++)^^ buff
f-1	01F0	65	72	5B	69	5D	20	3D	20	62	75	66	66	65	72	5B	69	er[i] = buffer[i
f-1	0200	5D	20	5E	20	63	6F	64	65	5B	69	20	25	20	6B	65	79	]^code[i % key
f-1	0210	6C	65	6E	67	74	68	5D	3B	0D	0A	7D	0D	0A	0D	0A	1A	length];^^}^^^^^
f-1	0220	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	^^^^^^^^^^^^^^^^
f-1	0230	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	^^^^^^^^^^^^^^^^
f-1	0240	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	^^^^^^^^^^^^^^^^
f-1	0250	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	^^^^^^^^^^^^^^^^
f-1	0260	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	^^^^^^^^^^^^^^^^
f-1	0270	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	1A	^^^^^^^^^^^^^^^^

Table 1

icing on the cake; however it completely reverses the decryption method. If the original run string was

cypher file.txt file.new ABRAHAM -2 LINCOLN -3

then the decryption run string would be

cypher file.new file.doc -3 LINCOLN -2 ABRAHAM

Fv.c—Listing Five

As I mentioned earlier, this is my replacement for the CP/M dump utility. It allows the user to pass one or two files in the run string for display. If one file name is passed in the run string, the output appears much like the CP/M dump.com output with the addition of the ASCII display. If two file names are passed, the output consists of a line from file 1, a line from file 2, and a third line containing the exclusive-ORing of the two files (labeled "dif"). In all cases, nonprintable characters are replaced with a caret (^) in the ASCII portion and nulls are replaced with an equal sign (=) to readily identify comparisons between two files. The comparative output is purely a byte-for-byte operation, and no attempt is made to realign the file to comparing characters as in a *compare* utility. The first file length controls display length. Table 1, page 60, shows an example of screen output from the run string *<fv cypher1.c>*, and Table 2, below, shows one from the run string *<fv cypher1.c cypher2.c>*.

Fstat.c—Listing Six

Descriptive statistics is the name of the game here, and as with any statistical evaluation, you must be brutally honest (at least with yourself) to draw an objective conclusion. The entire file is read, 16K at a time. As you read, the occurrences of each of the 256 tokens are accumulated and you obtain the sum of all bytes as well as the *min* and *max* token occurrences. The sum is divided by the total characters to obtain the mean, the

f-1	0000	2F	2A	09	63	79	70	68	65	72	31	2E	63	09	43	79	70	/*^cypher1.c^Cyp
f-2	0000	2F	2A	09	63	79	70	68	65	72	32	2E	63	09	43	79	70	/*^cypher2.c^Cyp
dif	0000	00	00	00	00	00	00	00	00	00	03	00	00	00	00	00	00	=======^=======
f-1	0010	68	65	72	20	6D	6F	64	75	6C	65	09	09	62	79	20	46	her module^^by F
f-2	0010	68	65	72	20	6D	6F	64	75	6C	65	09	09	62	79	20	46	her module^^by F
dif	0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	=============
f-1	0020	2E	41	2E	53	63	61	63	63	68	69	74	74	69	0D	0A	2A	.A.Scacchitti^^*
f-2	0020	2E	41	2E	53	63	61	63	63	68	69	74	74	69	0D	0A	2A	.A.Scacchitti^^*
dif	0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	=============
f-1	0030	2A	09	09	09	09	09	09	09	31	30	2F	31	30	2F	38	35	*^^^^^^^10/10/85
f-2	0030	2A	09	09	09	09	09	09	09	31	30	2F	31	31	2F	38	35	*^^^^^^^10/11/85
dif	0030	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	00	============^===
f-1	0040	0D	0A	2A	2A	0D	0A	2A	2A	09	53	69	6D	70	6C	65	20	^^**^^**^Simple
f-2	0040	0D	0A	2A	2A	0D	0A	2A	2A	09	43	6F	6D	70	6C	65	78	^^**^^**^Complex
dif	0040	00	00	00	00	00	00	00	00	00	10	06	00	00	00	00	58	========^^=====X
f-1	0050	63	79	70	68	65	72	20	6D	6F	64	75	6C	65	20	2D	20	cypher module -
f-2	0050	20	63	79	70	68	65	72	20	6D	6F	64	75	6C	65	20	2D	cypher module -
dif	0050	43	1A	09	18	0D	17	52	4D	02	0B	11	19	09	45	0D	0D	C^^^^^RM^^^^^^E^^
f-1	0060	65	6E	63	6F	64	65	73	20	64	69	72	65	63	74	6C	79	encodes directly
f-2	0060	20	67	65	6E	65	72	61	74	65	73	20	61	20	6B	65	79	generates a key
dif	0060	45	09	06	01	01	17	12	54	01	1A	52	04	43	1F	09	00	E^^^^^^^T^^R^C^^=
f-1	0070	20	77	69	74	68	20	75	73	65	72	20	6B	65	79	73	0D	with user keys^
f-2	0070	20	6F	66	20	73	6F	6D	65	20	70	72	69	6D	65	20	6C	of some prime l
dif	0070	00	18	0F	54	1B	4F	18	16	45	02	52	02	08	1C	53	61	=^^T^O^^E^R^^^Sa
f-1	0080	0A	2A	2A	0D	0A	2A	2F	0D	0A	0D	0A	23	69	6E	63	6C	^**^^*/^^^^#incl
f-2	0080	65	6E	67	74	68	0D	0A	2A	2A	20	20	20	20	20	20	20	ength^^**
dif	0080	6F	44	4D	79	62	27	25	27	20	2D	2A	03	49	4E	43	4C	oDMyb'%' -*^INCL
f-1	0090	75	64	65	20	3C	73	74	64	69	6F	2E	68	3E	0D	0A	0D	ude <stdio.h>^^^
f-2	0090	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
dif	0090	55	44	45	00	1C	53	54	44	49	4F	0E	48	1E	2D	2A	2D	UDE=^STDIO^H^-*-

Table 2

(continued from page 61)

max becomes the mode, and the range is the difference between min and max. Next, the 256-byte array is copied to a second array and sorted to obtain the median.

With all calculations completed, the numerical values of occurrences are displayed in a 16 × 16 display for evaluation. The statistical characteristics are displayed, and the program pauses to await some keyboard entry to display the histogram. Depressing the space bar prints a scaled horizontal histogram of 16 groups (0–15, 16–31, . . ., 241–256).

The ideal random file (which is what you want to see) would have the following characteristics:

mean	127.5
mode	not critical
range	< 20% of the total bytes divided by 256
median	at midpoint of range
histogram	reasonably flat

Remember I said "ideal!" A sequential file will display these ideal characteristics as well as a true random file. Also, files that look too good statistically should be just as suspect as those that don't. Table 3, below, is the output produced by this article "as is," and Table 4, page 63, is the output when it's encrypted with the run string

cypher crypt-tb.art new frederick -1 angelo -2 scacchitti -3

In all fairness, I must state that other, possibly better, statistical methods exist for determining randomness. I opted to use descriptive statistics because they are more easily understood and implemented.

Makef.c—Listing Seven
This is a simple enough utility but an absolute necessity if you are to evaluate encryption schemes. A file name of n 256-byte blocks is created, and if a value between 0 and 255 is passed, the file will be filled with this value.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	333	0	0	333	0	0
0	0	0	0	0	0	0	0	0	0	30	0	0	0	0	0
3249	1	6	17	0	1	0	15	39	39	0	1	85	50	162	
9	17	35	25	10	9	16	20	5	6	11	1	0	3	2	2
0	0	46	6	42	10	19	13	8	13	68	2	10	28	25	24
13	20	0	14	28	59	3	1	5	1	7	1	3	0	3	1
0	0	841	145	464	410	1528	315	217	551	914	6	75	518	337	749
766	287	14	683	832	1141	304	89	151	36	244	8	0	0	00	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
16640 characters read from file CRYPT-TB.ART
file mean = 85 06480/16640          mode = 32 ( 20 hex)
file median = 0 file range = 3250 [ min = 0 max = 3249 ]
scale = 156

  0 to  15 =  666 ¦¦*****
 16 to  31 =   30 ¦¦*
 32 to  47 = 3684 ¦¦************************
 48 to  63 =  162 ¦¦**
 64 to  79 =  327 ¦¦***
 80 to  95 =  146 ¦¦*
 96 to 111 = 7836 ¦¦***************************************************
112 to 127 = 3789 ¦¦*************************
128 to 143 =    0 ¦¦
144 to 159 =    0 ¦¦
160 to 175 =    0 ¦¦
176 to 191 =    0 ¦¦
192 to 207 =    0 ¦¦
208 to 223 =    0 ¦¦
224 to 239 =    0 ¦¦
240 to 255 =    0 ¦¦
```

Table 3

If no value (or one that is nonnumeric) is passed, each block contains a sequential count from 0 to 255. A benefit of this program is its ability to clean a disk. The user simply creates a file the size of the remaining disk space and then erases it. This results in all free disk space being set as the user defined.

Sp.c—Listing Eight

Along with the *fstat* utility, sp.c confirms or denies (maybe even questions) the statistical data you have received. A brute-force search method is used to find matching character strings in the file. By default, the search starts at the first character and searches the first 128 bytes for a match of four or more characters. If the match depth exceeds the block size it is searching, the program will return to CP/M after displaying the match data. If not, it will continue in its search. Block size to search, minimum depth of comparison, and starting point in the buffer may optionally be changed in the run string.

Also, if an additional argument is passed (one more than the three mentioned), the software converts the data in the buffer to delta form—that is, *element[n] = element[n] - element[n+1]* for all data in the buffer. After the conversion is made, the search scheme continues as before. This feature allows you to evaluate the file for some mathematical sequence.

One drawback to this program as it stands is the limiting factor of the buffer size. No attempt is made to search beyond it. This shouldn't matter for most text files.

Small-C, CP/M, and Miscellaneous

The function *chkkbd( )* enables you to stop display (program) activity if Ctrl-S is depressed. Following this with a Ctrl-C causes a return to CP/M, and any other character will allow the program to continue. This function calls a *bdos( )* function, so the user may have to modify this for other operating systems. The *getchx( )* function is a nonechoing

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
68	57	68	63	77	71	84	63	72	59	56	66	55	61	71	71
67	59	66	72	72	57	63	69	60	58	62	66	82	60	72	69
57	69	76	65	65	75	70	56	73	84	71	65	65	65	65	71
76	55	69	55	50	71	69	84	71	56	69	74	55	74	49	82
71	58	79	52	68	72	63	47	65	57	52	68	62	57	53	66
61	70	54	62	60	45	62	79	59	59	66	61	71	67	41	73
78	70	60	78	65	50	76	70	58	66	72	56	70	65	66	63
63	56	73	68	49	62	68	60	74	84	70	66	67	66	73	77
70	60	63	70	66	58	67	57	64	62	68	55	61	60	85	70
64	60	62	58	59	74	72	61	64	65	66	62	57	70	80	75
56	55	58	71	58	65	57	74	51	67	61	76	77	63	60	66
71	71	52	67	54	66	66	76	78	61	58	63	52	58	62	71
76	63	71	62	65	68	67	65	75	66	66	57	67	56	71	54
53	70	69	64	54	55	69	70	60	61	70	62	73	68	51	66
76	52	69	56	54	62	61	83	72	75	52	56	74	72	77	66
63	61	64	62	69	58	57	55	67	68	64	65	63	59	70	62

16640 characters read from file NEW
file mean = 126 07580/16640 mode = 142 (8E hex)
file median = 65 file range = 45 [min = 41 max = 85]
scale = 21

```
  0 to  15 = 1062 ||***************************************************
 16 to  31 = 1054 ||***************************************************
 32 to  47 = 1092 ||****************************************************
 48 to  63 = 1059 ||***************************************************
 64 to  79 =  990 ||************************************************
 80 to  95 =  990 ||************************************************
 96 to 111 = 1063 ||***************************************************
112 to 127 = 1076 ||****************************************************
128 to 143 = 1036 ||**************************************************
144 to 159 = 1049 ||**************************************************
160 to 175 = 1015 ||*************************************************
176 to 191 = 1026 ||*************************************************
192 to 207 = 1049 ||**************************************************
208 to 223 = 1015 ||*************************************************
224 to 239 = 1057 ||***************************************************
240 to 255 = 1007 ||************************************************
```

Table 4

(continued from page 63)

version of *getchar( )* that uses BDOS function 6. You can substitute *getchar( )* for *getchx( )*.

The functions *calloc( )*, *malloc( )*, and *cfree( )* are used for the dynamic allocation and deallocation of memory. My allocation/deallocation scheme is of the simple variety in which the programmer must pay heed to order or pay the consequences. The source code contained here should work with most implementations of these functions.

Printer output can be obtained from any of the programs by using the CP/M Ctrl-P function. It was the simplest method to implement.

Math functions (especially floating point) are difficult for Small-C. There are several routines in the fstat.c source that perform the necessary long and fractional calculations. It's not necessary to change these; however, if your compiler supports floats and longs, have at it.

Each program will display the usage if entered without the proper number of arguments in the run string. Also, because most software users begin to feel uncomfortable when their computer is off somewhere performing exotic calculations, each program displays status to the screen to put these fears at rest.

Cypher Benchmarks

My version, written in Small-C, is generic enough to adapt to any C compiler. Running on a 4-MHz, Z80-based CP/M system, it benchmarks at less than 1K per second for file I/O, 16K per second for file transposition, and approximately 4K per second per key for encryption. Key encryption is difficult to benchmark because it includes the time to generate the prime-length key, which varies from 1,009 to 1,999 characters in length.

And Finally

These tools should be employed with a measure of common sense. A strong cypher is indicated only when both statistically and patternwise indicated. (And it doesn't hurt to view the file either.) My intent in developing these utilities was to provide the cryptographer-programmer with a means to evaluate the strength of an encryption scheme as well as the resistance of schemes to cracking. Like most tools, however, these can be used for destructive as well as constructive purposes. The author assumes no liability for illegal use of these tools and sincerely believes they can result only in the development of stronger encryption schemes.

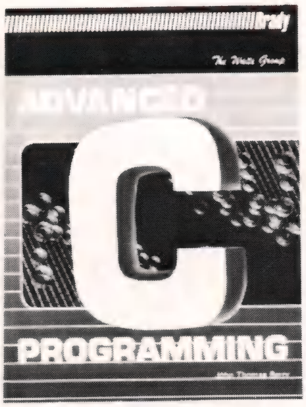
DDJ

(Listings begin on page 94.)

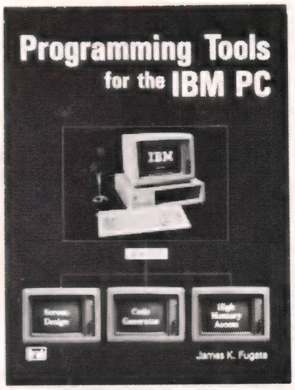
Vote for your favorite feature/article.
Circle Reader Service **No. 7**.

BRADY Knows Programming.

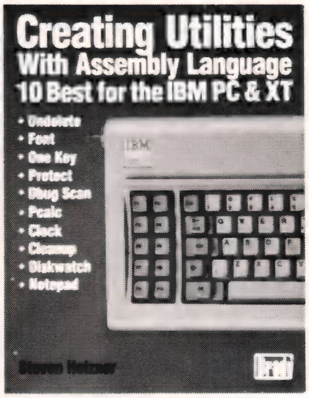
You'll learn to whip every element of your programming into shape with the latest information and guidance by America's foremost technical experts. Just call toll-free or use the coupon below to order today.



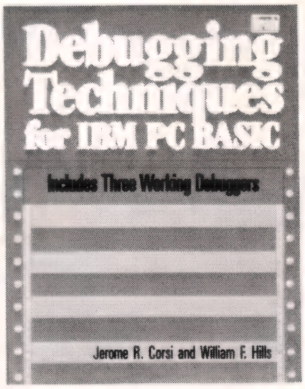
1. Beyond the basics, this guide explores new structured concepts to analyze and solve problems in C with tools of modularity, input and output functions, arrays and structures, and bit manipulation. $21.95



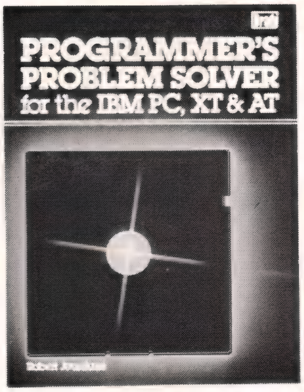
2. Includes a listing for the remarkable QUICKCODE code generator which automatically writes input, locate, and print statements. Also unleashes high memory access and makes screen design a cinch. $19.95



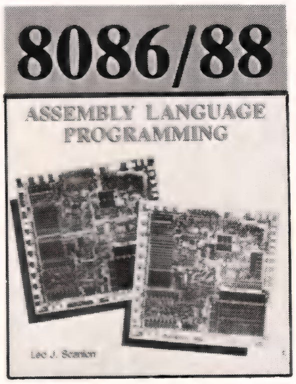
3. Learn the techniques used for creating assembly language utilities. Includes 10 of the most popular utilities such as DBUG SCAN, CLOCK, UNDELETE, ONE KEY, PCALC calculator and notepad and five others. $21.95 (Disk available)



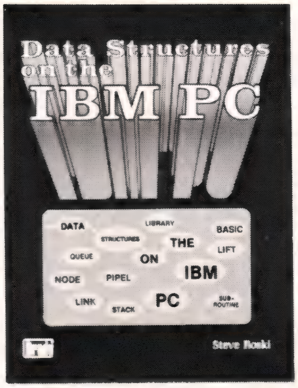
4. Includes code listings for three working debuggers including single-stepping, cross referencing, and mapping utilities. $19.95 (Disk available)



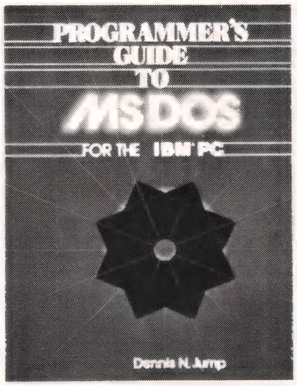
5. A definitive reference text for advanced programmers. You'll find over 150 discussions of common hardware-control tasks (in BASIC, Pascal, or C) as well as assembler overlays, drivers, and real-time operators. $22.95



6. Probes the inner workings of the 8086 (used by the AT&T 6300) and 8088 (IBM PC) chips...and describes specific techniques for using the full capability of these chip designs while programming in assembler. $18.95



7. Here's a compendium of many of the most useful, but often neglected, advanced programming concepts. A tutorial in format that uses BASIC for examples, it covers techniques such as: linked data structures; recursion; pipelining; and dynamic storage allocation. Includes listings for 25 subroutines. $21.95 (Disk available)



8. A breakthrough explanation to the technical operations of DOS for programmers. Shows how to use the I/O services with discussions of: character and non-character functions; directory and file management routines; and memory management. Includes BIOS functions and info on IBM "compatibles." $16.95

Now at your book or computer store. Or order toll-free today: **800-624-0023** In New Jersey: 800-624-0024

BRADY COMMUNICATIONS COMPANY, INC.
c/o Prentice Hall
P.O. Box 512, W. Nyack, NY 10994

Circle the numbers of the titles you want below. (Payment must be enclosed; or, use your charge card.) Add $1.50 for postage and handling. Enclosed is check for $_____ or charge to

☐ MasterCard ☐ VISA

Acc't # __________ Exp. date ______

Signature __________

Name __________

Address __________

City __________ State ______ Zip ______

(New Jersey residents, please add applicable sales tax.)

Dept. 3

GR-BKDD-BK(6)

1 (0-89303-473-8)	2 (0-89303-784-2)	3 (0-89303-584-X)	4 (0-89303-587-4)
5 (0-89303-787-7)	6 (0-89303-424-X)	7 (0-89303-481-9)	8 (0-8359-5655-5)

Brady

Circle **no. 219** on reader service card.

May PforCe™ Be With You!

NEW! Writing in C? Half your job could be done already. With PforCe. The first library of object-oriented C functions and subsystems. Written in C. Fully integrated, optimized, debugged, and ready to go.

High level functions to manipulate objects like windows. Fields. Screens. Menus. Change an object's characteristics globally. Or tailor them to a specific application. Low level functions for complete hardware control. Sophisticated subsystems to handle complex tasks. A database system with demand paging and B-trees.

But that's not all. PforCe's routines are easy to find and use Alphabetically. By funtional group. Or, while you're editing, through a pop-up utility. PforCe is available for Microsoft®, Lattice®, CI86™, and Wizard® compilers. And PforCe comes with an easy to follow tutorial to help you become more productive quickly. **$475.** Order before June 15, 1986 and save $80. **Special offer $395.**

Plink™86 Plus Adds A New Dimension In Modular Programming.

NEW! You can create up to 4,095 overlays, in one or many files, on one or many disks. And in any MS-DOS/PC DOS environment Plink 86 Plus will automatically cache overlays using all or part of the available memory including the IBM AT extended memory. Allowing you to get better performance from your large machines.

Merge object modules so you can package them together. Reload overlays upon function return. Allocate library modules to overlay structures, automatically.

Use the same module in different programs. Change the overlay structure of an existing program without recompiling. Or, use one overlay to access code and data in other overlays. **$495.**

Pfix™86 Plus. The Most Advanced Symbolic Debugger On The Market Today.

NEW! Debug without a listing since you can see and enter symbolic names or absolute addresses in breakpoints, data displays, expressions, or with the in-line assembler. Configure your own menus and keys. Get tracebacks to save trace histories. Set breakpoints in the source file window. Stop and restart a debugging session at any time. And, display source files, disassembled object, data area, stack, breakpoint settings, CPU and coprocessor registers, simultaneously. **$395.**

Pre-C.™ The Lint For MS-DOS.

NEW! Find usage errors almost impossible to detect with a compiler. Cross-check multiple source files and parameters passed to functions. Uncover interface bugs that are difficult to isolate. All in a single pass.

Our new Pre-C gives you the full range of features C programmers working in UNIX® have come to expect from their Lint program analyzer.

Capabilities no C compiler, with or without program analyzing utilities, can offer. In fact, Pre-C outlints Lint, since you can handle analyses incrementally. **$395.**

Pfinish™ Maximizes Your Program's Efficiency.

Pfinish helps you turn your beta-test product into a software work of art. It analyzes your program or the entire operating environment during execution, and produces reports which tell you how much time was spent in each routine or interrupt, who called it, how many times, and much more.

Wasteful, inefficient, or non-optimal areas of code become immediately apparent, whether they're in your code, the compiler library, or in the operating system routines themselves. **$395.**

Ptel™ Gets The Lead Out Of Binary File Transfer.

NEW! Ptel is the first file transfer software to include an on-line service that gives you immediate electronic updating of your entire Phoenix software library. New software bulletins. Sample Files. On-line technical support. And, an opportunity to exchange information with other Phoenix software users.

Plus, you get error-free file transfer and access to mainframes, minis, and micros. ASCII. XModem. Modem7. Telink and Kermit. You can transfer 8-bit binary files over a 7-bit data path with Kermit. Transfer groups of files using ambiguous file references. Or, transfer whole subdirectories with a single command using Telink. Ptel keeps track of the original file size and creation date.

Ptel also offers a script language, backward scrolling, and the ability to handle DOS commands from inside the program. **$195.**

Spoil Yourself. Get A Pfantasy Pac™ Today!

NEW! Get six of the best MS-DOS programming packages on the market today. The new Plink86 Plus, and the latest versions of Pfix86 Plus, Ptel, Pmate, Pmaker, and Pfinish. A **$1900** value. Plus, the new Ptel on-line software update service. All for **$1295.**

Get A 15 Percent Discount On All Software Purchased With A Pfaster™286 Board.

NEW! Now you can increase the performance of your IBM PC, XT or compatible, and make money doing it. Buy a 1Mb, 2Mb, or with our new PfasterAccess™/2Mb option, a 4Mb accelerator board that supports EGA, and we'll give you 15 percent off any Phoenix program development tools that you buy at the same time. No dealers please. This offer is good for a limited time.

Programmers, make your programming dreams come true. Call or write:
Phoenix Computer Products Corporation
320 Norwood Park South
Norwood, MA 02062 (800) 344-7200.
In Massachusetts (617) 762-5030

Programmers' Pfantasies by

Phoenix

Listing One (Text begins on page 8.)

```
/**
**
**      LJ.C  -- A printing utility for the HP LaserJet
**
**         This program prints a series of files on the LaserJet
**         printer.  The files are printed in a ''landscape'' font at
**         17 characters to the inch.  To take advantage of this
**         density, two ''pages'' of information from the file are
**         printed on each piece of paper (left and right halves).
**
**         Usage is: LJ file1 file2 file3 ...
**
**         Where file# is a valid MS-DOS filename, included on the
**         command line.  This program is compatible with Lattice C
**         on the IBM PC and the HP Touchscreen computers.
**
**         Joe Barnhart         original version      May 5, 1985
**         Ray Duncandate and time stamping          May 22, 1985
**         Joe Barnhart         revised date stamping          June 6, 1985
**         Ray Moon  modified for CI86    December 13, 1985
**                              & revised EOF test
**
**/

#define CI86         1          /* Remove this #define => Lattice C version */

#ifdef CI86
#include <stdio.h>
#else
#include <h\stdio.h>
#endif
#define MAXLINE 56                      /* maximum lines per page */
#define Page         '\f'                       /* for compilers without '\f' */
#define TAB          8                          /* width of one tab stop */

#ifdef CI86
typedef struct {
                unsigned short ax,bx,cx,dx,si,di,ds,es;
        } REGSET;
#else
typedef struct {
                int ax, bx, cx, dx, si, di;
        } REGSET;
#endif

main(argc, argv)
        int argc;
        char *argv[];
{
        int filenum;
        FILE *fp, *prn, *fopen();

        if( (prn = fopen(''PRN:'', ''w'') ) == NULL )
                printf(''Error opening printer as file.\n'');
        else {
                /* initialize the LaserJet for landscape printing */
                fprintf(prn, ''\033E\033&l1o\033(s17H\033&l8d6E'' );
                for(filenul=1; filenum < argc; filenum++) {
                        fp = fopen(argv[filenum], ''r'');
                        if (fp == NULL)
                                printf(''file %s doesn't exist.\n'', argv [filenum])
                        else {
                                printf(''Now printing %s\n'', argv[filenum]);
                                printfile(fp, prn, argv[filenum]);
                                fclose(fp);
                        }
                ~
                fprintf(prn, ''\015\033E'');   /* clear LaserJet */
        }
}

printfile(fp,prn,filename)
        FILE *fp,*prn;
        char *filename;
{
        int pagenum = 1;
        while( !feof(fp)) {
                fprintf(prn, ''\033&a0r85m5L\015'');     /* set left half */
                printpage(fp,prn);                        /* print page */
                if( !feof(fp)) {                          /*if more .. */
                        fprintf(prn, ''\033&a0r171m91L''); /*set right half */
                        printpage(fp,prn);                /* print another */

                }
                stamp(prn, filename, pagenum++);          /* title */
                fputc(PAGE, prn);                         /* kick paper */
        }
}

printpage(fp,prn)
        FILE *fp, *prn;
{
        int c,line,col;

        line = col = 0;
        while(line < MAXLINE)
                switch(c = fgetc(fp)) {
                        case '\n':                        /* newline found */
                                col = 0;                  /* zero column */
                                line++;                           /* adv line cnt */
                                fputc('\n',prn);
                                break;
                        case '\t':                        /* TAB found */
                                do
                                        fputc('\040',prn);
                                while ((++col % TAB) != 0);
                                break;
```

(continued on page 70)

Listing One (Listing continued, text begins on page 8.)

```
                    case PAGE:                    /* page break or */


                    case EOF:                     /* EOF found */
                            line = MAXLINE;               /* force terminate */
                            break;
                    default:                      /* no special case */
                            fputc(c,prn);                 /* print character */
                            col++;
                            break;
                }
}

stamp(prn, filename,pagenum)
        FILE *prn;
        char *filename;
        int pagenum;
{
        char datestr[10], timestr[10];

        fprintf(prn, ``\033&a51171M'');                   /* widen margins */
        fprintf(prn, ``\015\033&a58R'');          /* move to row 58 */
        fprintf(prn, ``File:  %-113s'', filename);
        fprintf(prn, ``Page %-3d'', pagenum);
        timestamp(timestr);
        datestamp(datestr);
        fprintf(prn, ``   %s    %s'', datestr, timestr);
}

datestamp(datestr)
        char *datestr;
{
        REGSET regs;
        int month, day, year;

        regs.ax = 0x2a00;
#ifdef CI86
        sysint21(&regs,&regs);
#else
        int86(0x21,&regs,&regs);
#endif
        month = (regs.dx >> 8) & 255;
        day = regs.dx & 255;
        year = regs.cx - 1900;
        sprintf(datestr, ``%02d/%02d/%02d'', month, day, year);
}

timestamp(timestr)
        char *timestr;
{
        REGSET regs;
        int hours,mins;

        regs.ax = 0x2c00;
#ifdef CI86
        sysint21(&regs,&regs);
#else
        int86(0x21,&regs,&regs);
#endif
        hours = (regs.cx >> 8) & 255;
        mins = regs.cx & 255;
        sprintf(timestr, ``%02d:%02d'', hours, mins);

}
```

End Listing

PROBLEM: Handling your need for more megabytes, without spending megabucks on a new drive.

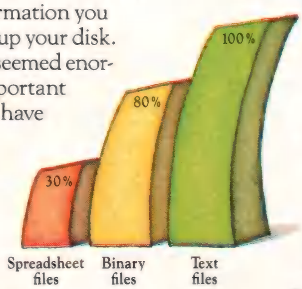
THE SOFTLOGIC SOLUTION: Cubit™

Now get up to twice the capacity from all your storage media.

You know what happens. The more you use your computer, the more information you create. And the faster you fill up your disk.

The 10MB drive that once seemed enormous is now jammed with important files. That 20MB that should have lasted years is crowded in a matter of months.

Of course you could keep buying bigger hard drives. Or you could get Cubit and get the maximum storage space from the drives you already have.



With Cubit, you'll get as much as 100% compression on data files, effectively doubling the storage capacity of all your magnetic media.

What is Cubit?

In brief, Cubit is an advanced software tool that automatically reduces the number of bytes required to store a file, then converts the file back to its original size when retrieved. Some programmers call this effect "data compression," others, "disk expansion." Either way, the result is the same.

Here's how it works. When Cubit compresses a file, it first compares each word to its massive English word dictionary. Words that match are reduced to a predetermined code of just one, two or three bytes each. It then saves the abbreviated version to disk. Decompression works just the opposite.

To accommodate other words and symbols, Cubit uses two more compression techniques. One assigns new, shorter codes to unusual words. Another compresses according to the frequency of character strings in non-text data. So no matter what kind of files you create, Cubit ensures maximum space savings.

Best of all, you'll be using the same fast, reliable data compression techniques used on mainframe computers for decades.

How much disk space will you save?

Because the vast majority of data created on PC's is standard ASCII text—letters, numbers and other English language symbols—we've optimized Cubit for word processing and database files. With these, you'll get a minimum of 50% expansion on up to a full 100% or more.

At the same time, you can expect a significant 30% to 50% improvement with other kinds of data. Including spreadsheet files, program code, graph and image files, even binary data.

And Cubit works just as well with floppies and tape cassettes as it does with hard disk drives.

Run Cubit where you want, when you want.

Maybe you'll want to use Cubit for all your files, or maybe just some. So Cubit lets you specify exactly which files to work on and which ones to leave alone.

In RAM resident mode, Cubit works quickly and invisibly, compressing and decompressing right from within any program you run. Or use Cubit's powerful file management mode. It supports wild-card and global file names, and addresses sub-directories up to thirty levels deep.

Save time and money, as well as disk space.

A compressed file is a smaller file. So with Cubit, back-ups take less time, as well as less space. And communicating compressed files means significant savings on phone line charges.

Any way you look at it, Cubit will pay for itself in no time. And that's especially true now.

Special limited time offer.

Buy Cubit now and you'll save even more. Because for a limited time, you can buy Cubit at the special introductory price of just $49.95.* But hurry. This special price won't last long.

Ask for Cubit at your computer dealer. Or order directly from SoftLogic Solutions by calling 800-272-9900 (603-627-9900 in NH), or mail in the coupon below.

Special pricing is available when you buy Cubit along with other SoftLogic products including DoubleDOS, Software Carousel and Disk Optimizer. Ask for details.

Cubit™ $49.95*

YES! Please send me____copies of Cubit at this special introductory price.

Name________________

Company________________

Address________________

City________ State/Zip________

Check Enclosed ☐ VISA ☐ MC ☐ AMEX ☐

Card # Exp. Date

Signature________________

SoftLogic Solutions, Inc.
530 Chestnut Street
Manchester, NH 03101
800-272-9900
(603-627-9900 in NH)

SOFTLOGIC SOLUTIONS

Call today: 800-272-9900

*plus $5.00 shipping and handling.

Circle **no. 293** on reader service card.

Listing One **(Text begins on page 18.)**

```
#include <stdio.h>
#include <dos.h>

#define NUMROWS         25
#define NUMCOLS         80
#define VIDBASE         0xb0000000  /* Base address of video screen
                                     * in canonical form.
                                     */

#define NORMAL          0x07    /* Basic Attributes. Only one    */
#define UNDERLINED      0x01    /* of these may be present.      */
#define REVERSE         0x70

#define BLINKING        0x80    /* May be ORed with the above    */
#define BOLD            0x08    /* and with each other           */

/*------------------------------------------------------------------*/

typedef struct
{
        char    letter;
        char    attribute;
}
CHARACTER;

typedef CHARACTER       DISPLAY[ NUMROWS ][ NUMCOLS ];

static DISPLAY far *Screen = (DISPLAY far *) VIDBASE;

/*------------------------------------------------------------------*/

static int      Row = 0;
static int      Col = 0;

/*------------------------------------------------------------------*/

static void     fix_cur()
{
        /*      Direct video writes won't actually move the cursor
         *      so we'll do that with a ROM-BIOS call. move_cur
         *      puts the cursor at Row, Col.
         */

        union REGS      Regs;

        Regs.h.dh = Row;
        Regs.h.dl = Col;
        Regs.h.bh = 0 ;                 /* Use "active" video page       */
        Regs.h.ah = 2 ;                 /* Function 2, set cursor position */
        int86( 0x10, &Regs, &Regs);     /* Video int                     */
}

/*------------------------------------------------------------------*/

void    setcur( row, col )
{
        Row = row;
        Col = col;
        fix_cur();
}

/*------------------------------------------------------------------*/

void    getcur( rowp, colp )
int     *rowp, *colp;
{
        *rowp = Row;
        *colp = Col;
}

/*------------------------------------------------------------------*/

void    d_putc( c, attrib )
{
        switch( c )
        {
        case '\r':      Col = 0;
                        break;

        case '\n':      if( ++Row >= NUMROWS )
                                Row = 0;
                        break;

        case '\b':      if( --Col < 0 )
                                Col = 0;
                        break;
        default:
                        (*Screen)[ Row ][ Col ].letter    = c      ;
                        (*Screen)[ Row ][ Col ].attribute = attrib ;
                        if( ++Col >= NUMCOLS )
                        {
                                Col = 0;
```

```
                                        if( ++Row >= NUMROWS )
                                                Row = 0;
                                }
                                break;
        }

        fix_cur();
}

/*----------------------------------------------------------------------*/

void        d_puts( str, attrib )
register char   *str;
register int    attrib;
{
        while( *str )
                d_putc( *str++, attrib );
}

/*----------------------------------------------------------------------*/

void        clrs( attrib )
{
        /*      Clears the screen. The cursor is not moved.
        */

        CHARACTER far   *p = (CHARACTER *)VIDBASE ;
        register int    i ;

        for( i = NUMROWS * NUMCOLS;  --i >= 0 ; )
        {
                (p  )->letter    = ' ';
                (p++)->attribute = attrib ;
        }
}

/*----------------------------------------------------------------------*/

main()
{
        clrs  ( NORMAL );
        setcur( 0, 0 );

        d_puts("Normal\n\r",                      NORMAL                   );
        d_puts("Normal blinking\n\r",             NORMAL|BLINKING          );
        d_puts("Reverse\n\r",                     REVERSE                  );
        d_puts("Underlined\n\r",                  UNDERLINED               );
        d_puts("Underlined bold\n\r",             UNDERLINED|BOLD          );
        d_puts("Underlined blinking bold\n\r",    UNDERLINED|BOLD|BLINKING);

        exit(0);
}
```

End Listing One

Listing Two

```
| main()                                                          |
| {                                                               |
|        /* Exit status == 0 if SHLEV environment exists, else    |
|         * exit status == 1                                       |
|         */                                                       |
|                                                                 |
|        exit( getenv("SHLEV") == 0 );                            |
| }                                                               |
|                                                                 |
```

End Listing Two

LISTING THREE

```
+
|
+
|                                                               |
|       #                                                       |
|       shlev                                                   |
|       if( $status ) then                                      |
|               setenv SHLEV 1                                  |
|       else                                                    |
|               setenv SHLEV "`expr $SHLEV + 1`"                |
|       endif                                                   |
|       #                                                       |
|       set prompt="[$SHLEV,!] "                                |

|
```

End Listings

Listing One **(Text begins on page 42.)**

```
/*
 *  SINE.C
 *
 *  Plot sin(x) vs. x on an enhanced color display with
 *  an IBM enhanced graphics adapter in the high resolution mode.
 *  Purpose is to test the video routines.
 *
 *  by Nabajyoti Barkakati, Silver Spring, MD 20904.
 *
 */

#include <stdio.h>
#include <math.h>

#define BLUE    1                        /* Color number 1 is BLUE      */
#define RED     4                        /* Color number 4 is RED       */
#define BOTTOM 4                         /* Bottom margin               */
#define TOP     345                      /* Top margin                  */
#define LEFT    4                        /* Left margin                 */
#define RIGHT   635                      /* Right margin                */
#define TWOPI   6.283                    /* Approximate value of 2 Pi   */
#define MAXPNT 100                       /* Points on the sinusoid      */

main()
{
    int i, x, y, oldx, oldy, midpoint;
    double xd, yd, ampl;

    v_init (RED);                        /* Initialize the display      */

    midpoint = (TOP - BOTTOM)/2;
    ampl = (double)midpoint - (double)BOTTOM;

    oldx = LEFT;
    oldy = midpoint;

    for (i=0; i<=MAXPNT; i++)
    {
        yd = ampl * sin(TWOPI * ((double)i)/((double)MAXPNT));
        xd = ((double)RIGHT - (double)LEFT)* (double)i / (double)MAXPNT;
        x  = LEFT + (int)xd;
        y  = midpoint + (int)yd;

/* Draw a line from the old point to the new point */

        v_draw (oldx, oldy, x, y, BLUE);

/* Save the new coordinates */

        oldx = x;
        oldy = y;
    }

/* Draw a box around the plot */

    v_draw (LEFT, BOTTOM, RIGHT, BOTTOM, BLUE);
    v_draw (RIGHT, BOTTOM, RIGHT, TOP, BLUE);
    v_draw (RIGHT, TOP, LEFT, TOP, BLUE);
    v_draw (LEFT, TOP, LEFT, BOTTOM, BLUE);

/* Done */

}
```

End Listing One

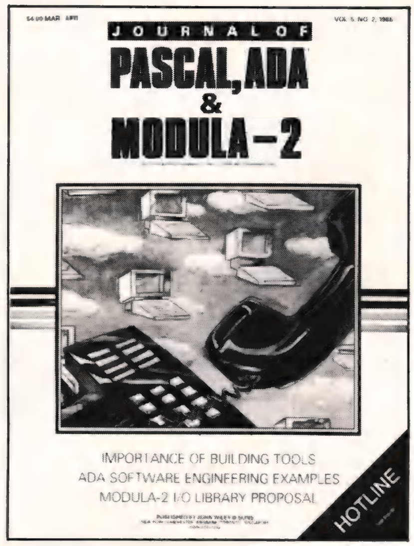
Listing Two

```
/*
 *  VIDEO.C
 *
 *  This file contains the video display modules. Uses the int86
 *  function of Lattice C 2.14 to draw graphs on an enhanced color
 *  display with the IBM enhanced graphics adapter.
 *
 *  by N. Barkakati, Silver Spring, MD 20904.
 *
 */

#include <dos.h>
#define void        int
#define EGAMODE     16                   /* EGA in high resolution      */
#define MAXROW      24
#define MAXCOL      79
#define MAXYDOT     349                  /* Max. columns and rows on    */
#define MAXXDOT     639                  /* enhanced color display      */

#define BIOS_VIDEO 16                    /* BIOS Video service int. no. */
#define SETMODE     0                    /* Service: set video mode     */
#define SETCOLOR    11                   /* Service: set color pallette */
#define WRITE_PIX   12                   /* Service: write pixel        */

static union REGS  xr,yr;                /* See dos.h for explanation   */

/*  v _ i n i t
 *
 *  Initialize the display. Put it in EGA hi-resolution mode.
 *  Set background color.
```

```
 *
 */
void v_init(bgcolor)
int bgcolor;
{
/*  ROM BIOS Video functions -- mode 16 is EGA in high-resolution
 *  (640x350 pixels)
 */
    xr.h.ah = SETMODE;
    xr.h.al = EGAMODE;
    int86 (BIOS_VIDEO, &xr, &yr);

/*  Set color.
 */
    xr.h.ah = SETCOLOR;
    xr.h.bh = 0;
    xr.h.bl = bgcolor;
    int86 (BIOS_VIDEO, &xr, &yr);

/* Done */
}
/*-------------------------------------------------------------------*/

/*
 *  v _ d r a w
 *
 *  Draw a line of specified color between the two points (x1,y1)
 *  and (x2,y2). Uses Bresenham's Algorithm described in:
 *  J.D. Foley and A. Van Dam, FUNDAMENTALS OF INTERACTIVE
 *  COMPUTER GRAPHICS, Addison-Wesley, 1982, pp.433-435.
 *
 */
void v_draw(x1, y1, x2, y2, color)
int x1, y1, x2, y2, color;
{
    int dx, dy, incr1, incr2, incr3, d, x, y, xend, yend;
    dx = abs(x2-x1);
    dy = abs(y2-y1);
    if (dy<=dx)
    {                  /* Absolute value of slope of line is less than 1 */
        if (x1>x2)
        {                    /* Start at point with smaller x coordinate */
            x = x2;
            y = y2;
            xend = x1;
            dy = y1-y2;
        }
        else
        {
            x = x1;
            y = y1;
            xend = x2;
            dy = y2-y1;
        }
        d = 2*dy-dx;
        incr1 = 2*dy;
        incr2 = 2*(dy-dx);
        incr3 = 2*(dy+dx);
        putdot (x, y, color);
        while (x < xend)
        {
            x += 1;
            if (d >= 0)
            {
                if (dy<=0)
                {                               /* Negative or zero slope */
                    d += incr1;
                }
                else
                {                                       /* Positive slope */
                    y += 1;
                    d += incr2;
                }
            }
            else
            {
                if (dy>=0)
                {                               /* Negative or zero slope */
                    d += incr1;
                }
                else
                {                                       /* Positive slope */
                y -= 1;
                d += incr3;
                }
            }
            putdot (x, y, color);
        }                                                   /* end while */
    }                                                       /* end if    */
    else                  /* Absolute value of slope is greater than 1 */
    {
        if (y1>y2)
        {                  /* Start with point with samller y coordinate */
            y = y2;
            x = x2;
            yend = y1;
            dx = x1-x2;
        }
        else
        {
            y = y1;
            x = x1;
            yend = y2;
        }
        d = 2*dx-dy;
        incr1 = 2*dx;
        incr2 = 2*(dx-dy);
        incr3 = 2*(dx+dy);
        putdot (x, y, color);
        while (y < yend)
        {
```

(continued on page 77)

How can you stay on the cutting edge of software development...for as little as 6¢ a day?



JOURNAL OF PASCAL, ADA & MODULA-2

Editor: **Dr. Richard S. Wiener,** *University of Colorado, Colorado Springs, Colorado*

System programming . . . graphics . . . numerical computing . . . operating systems . . . artificial intelligence . . . computer-aided instruction . . . embedded real-time systems . . . discrete-event simulation . . .

Just a part of today's tremendous excitement in computer science and software development.

And just a part of the many challenging areas explored by JOURNAL OF PASCAL, ADA & MODULA-2.

Within its pages, you'll find informative, peer-reviewed articles on the advanced capabilities of these three languages, written by top academic researchers and software development practitioners from government and large and small companies. You'll benefit from:

- new insights into the broader issues of the software development process in such articles as *Modular Software Construction and Object-Oriented Design Using Ada* and *Modular Software Construction and Object-Oriented Design Using Modula-2.*
- FREE utilities like *A Pascal Factor Analysis Program* and *Dynamic Multidimensional Arrays,* which offer interesting solutions and effective approaches to common programming and design problems.
- fresh solutions to graphics programs and new ideas you can use yourself, including *Real World and Standardized Graphics for Microcomputers* and *Pascal Graphics Routines for the Anderson Jacobson 232.*
- in-depth articles on the latest software systems such as *VMS Modula-2 Compiler Reviewed.*
- detailed implementation reports such as *Seven Modula-2 Compilers Reviewed* and *Using JANUS Ada.*

Plus critical reviews of the latest literature, a reader's forum, and a directory of upcoming seminars, conferences and educational opportunities.

All for as little as 6¢ a day

Individual subscribers to the JOURNAL OF PASCAL, ADA & MODULA-2 pay only $20 per year for six bimonthly issues — *a cost of just 6¢ a day.* A small price to pay for so much news on the latest developments that are important to you, your work, your profession.

Subscribe today!

Simply complete and mail the attached order form. Or call TOLL-FREE 1-800-526-5638. In New Jersey call collect 201-342-6707.

John Wiley & Sons
605 Third Avenue
New York, N.Y. 10158

ORDER FORM

Mail to: **John Wiley & Sons,** Subscription Dept. 6-0507
605 Third Avenue, New York, N.Y. 10158

YES! Please send me the JOURNAL OF PASCAL, ADA & MODULA-2 (ISSN 0735-1232), one full year of six bimonthly issues at the price listed below. I understand that if I am not completely satisfied with my first issue, I can cancel my subscription and receive a complete 100% refund. I keep my first issue, with Wiley's compliments.

Outside U.S. add $19 for surface postage & handling, or add $37 for airmail delivery.

Check one: ☐ Individual $20
☐ Institution $48

☐ Payment enclosed (U.S. currency only).
☐ Bill me.

Charge to my: ☐ MasterCard ☐ VISA ☐ American Express

Card # ____________ Exp. Date ________

Signature ____________

Name ____________

Company ____________

Address ____________

City/State/Zip ____________

Service begins upon receipt of payment.

For faster service call TOLL-FREE 1-800-526-5368. In New Jersey call collect 201-342-6707.

6-0990

Circle **no. 299** on reader service card.

Listing Two **(Listing continued, text begins on page 42.)**

```
            y += 1;
            if(d >= 0)
            {
                if (dx <= 0)
                {                                  /* Negative or zero slope */

                    d += incr1;
                }
                else
                {                                        /* Positive slope */
                    x += 1;
                    d += incr2;
                }
            }
            else
            {
                if (dx >= 0)
                {                                  /* Negative or zero slope */
                    d += incr1;
                }
                else
                {                                        /* Positive slope */
                    x -= 1;
                    d += incr3;
                }
            }
            putdot (x, y, color);
        }                                                   /* end while */
    }                                                       /* end else  */
/* Done */
}
/*-----------------------------------------------------------------*/

/*
 *  p u t d o t
 *
 *  Put a dot of specified color at location (x,y) on screen.
 *  Check if dot coordinates are within screen bounds.
 *
 *                      ^ y-axis
 *                      |
 *  Convention:         |_________|   Origin at lower left corner of screen.
 *                      |         |
 *           (0,0)->|_________| x axis
 *
 */
void putdot(x, y, color)
int x, y, color;
{
    if ( x<0 | x>MAXXDOT | y<0 | y>MAXYDOT ) return;

    xr.x.dx = MAXYDOT - y;
    xr.x.cx = x;
    xr.h.ah = WRITE_PIX;
    xr.h.al = color;
    int86 (BIOS_VIDEO, &xr, &yr);

 /* Done */

 }
```

End Listings

THE PROGRAMMER'S SHOP™

provides complete information, advice, guarantees and every product for Microcomputer Programming.

We support MSDOS (not just compatibles) PCDOS, Xenix-86, CPM-80, Macintosh, Atari ST, and Amiga.

SERVICE: FREE NEWSLETTER

Software development and AI on micros: trends, forecasts, controversies, innovations, and techniques. Plus an announcement of 80 NEW tools. **CALL** for the "Newsletter Packet."

RECENT DISCOVERY

dBrief, the dBASE Assistant - optional syntax directed editing, screen gen, graphics, speed coding. dBASE II, III, Clipper. PC $ 95

C Support-Systems

Basic C Library by C Source PC $139
C Debug - Source debuggers - by Complete Soft ($269), MSD ($149).
CPRINT - by ENSCO MS $ 45
C Sharp - well supported. Source, **realtime,** tasks MS $600
C ToolSet - DIFF, xref, source MS $135
Lattice Text Utilities MS $105
The HAMMER by OES Systems PC $179
PC LINT - Checker MS $125
SECURITY LIB - add encrypt to MS C. C86 programs. Source $250 PC $125

C-Screens, Windows, Graphics

Curses by Lattice PC $109
C Power Windows by Entelekon PC $119
C Windows by Syscom PC $ 89
ESSENTIAL GRAPHICS - fast, fonts, no royalties PC $219
GraphiC - source in C PC $219
Topview Toolbasket by Lattice PC $209
View Manager for C by Blaise PC $219
Windows for C - fast PC $149
Windows for Data - validation PC $259

DEBUGGERS

Advanced Trace-86 by Morgan Modify ASM code on fly. PC $149
CODESMITH - visual, modify and rewrite Assembler PC $129
C SPRITE - data structures PC $149
Periscope I - own 16K PC $269
Periscope II - symbolic, "Reset Box," 2 Screen PC $129
Pfix-86 Plus Symbolic Debugger by Phoenix - windows PC $289
Software Source by Atron - Lattice, MS C, Pascal, Windows single step, 2 screen, log file. MS $115
w/Breakswitch $199

Features

Panel Screen Generator - Create screen with editor, generates code. Full data validation, windows, no royalties. Specify Lattice, MSC, C86, MS Fortran or Pascal MS $239

Microsoft Cobol Tools - symbolic, windowing debugger w/source support. Plus cross reference. Menu Handler, mouse support. $310

Fortran & Supporting

ACS Time Series $469
Forlib + by Alpha - graphics and file routines, Comm. MS $ 59
MACFortran by Microsoft - full '77 Includes ASM output MAC $229
MS Fortran MS **$219**
No Limit - Fortran Scientific PC $129
PolyFortran - xref, pp, screen MS $149
Prospero - '66, reentrant MS $349
RM Fortran - enhanced "IBM Professional Fortran" MS $399
Scientific Subroutines - Matrix MS $149
Statistician by Alpha MS $269
Strings and Things - register, shell PC $ 59

MultiLanguage Support

BTRIEVE ISAM MS $199
BTRIEVE/N-multiuser MS $469
CODESIFTER - Execution PROFILER. Spot bottlenecks. Symbolic, automatic. MS $109
MultiHALO Graphics - Multiple video boards, printer, rich. Animation, engineering, business. Any MS language, Lattice, C86 PC $249
PLINK 86 - a program-independent overlay linker to 32 levels. MS $279
PLINK-86 PLUS - incremental MS $379
Pfinish Performance Analyzer PC $279
Profiler by DWB Associates MS $ 99
PolyLibrarian MS $ 85
PVCS Version Control MS $359
Screen Sculptor - slick, thorough, fast, BASIC, PASCAL. PC $ 99
ZAP Communications - VT 100, TEK 4010 emulation, full xfer. PC $ 85

TURBO PASCAL and SUPPORT

BORLAND: Turbo 3.0 $ 49
3.0 with 8087 or BCD $ 79
3.0 with 8087 and BCD $ 85
Turbo Graphix - graphs, windows $ 39
Turbo Toolbox or Editor $ 55
Turbo Tutor $ 29
TURBO . . . Asynch by Blaise, full $ 85
Power Tools by Blaise - library $ 85
Power Utilities - profiler, pp $ 85
Professional - interrupts, macros. $ 50
OTHERS: Screen Sculptor ($99), Pascal Pac ($100), Tidy ($45).

OTHER LANGUAGES

APL+/PC PC $469
CLIPPER dBASE Compiler MS $449
ED/ASM-86 by Oliver PC $ 85
HS/FORTH - '79 & '83 Standards, full RAM, ASM, BIOS, interrupts. Graph, multi-task, optimizer MS $250
MacASM - fast MS $ 99
MasterForth by MicroMotion - floating point and relocator extensions available: Call MAC or PC $125
Microsoft MASM - faster MS $109
Microsoft PASCAL MS $199
MODULA: M2SDS - popular PC $ 69
MICROTEC PASCAL - for performance: extensions like packages, "Iterators", 5 memory models. 65 bit 8087 strings. Space vs. speed MS $665
PASM - by Phoenix MS $219
Prospero Pascal - full ISO + MS $349
RPG II by Lattice PC $675
Turbo Edit/ASM - by Speedware PC $ 85

XENIX-86 & SUPPORT

Basic - by Microsoft $279
Cobol - by Microsoft $795
Fortran - by Microsoft $399
PANEL Screen LIB - multi-language $539
Xenix Complete Development System $985

OTHER PRODUCTS

dBASE to C Translator: dBx - no royalties, addon ISAM, Library **Pioneer it** MS $ 350
Source $1000
HTest/H Format - XT Fix PC $ 119
Microsoft Windows PC $ 75
Opt Tech Sort - sort, merge MS $ 85
Polymake by Polytron MS $ 85
PS MAKE - Directly execute or Gen a batch file, batch, interactive. MS $ 129
Qwik Net - critical path, 125 tasks resources, thorough; usable PC $ 316
SECRET DISK by Lattice PC $ 49
SET: SCIL MS $ 319
SoftEst - Softwear Estimating and reporting. Pioneer it. MS $ 350
Texsys - control source MS $ 89
Visible Computer: 8088 - Simulates demos or any .exe. com, Debugger. 350 pg. tutorial PC $ 59

Note: All prices subject to change without notice. Mention this ad. Some prices are specials. Ask about COD and POs. All formats available.

Call for a catalog, literature, advice and service you can trust

NEW HOURS
8:30 AM - 8:00 PM EST.

800-421-8006

THE PROGRAMMER'S SHOP™
128-D Rockland Street, Hanover, MA 02339
Mass: 800-442-8070 or 617-826-7531 386

"It's not often that I receive service from a company that I consider exceptional but you've managed to do it . . . I am impressed that you place the needs of your customers above making a "quick" profit and I hope to have the opportunity to do business with you again in the future . . ."
Samual Solon
White Pine Software, Inc.

Circle **no. 143** on reader service card.

Listing Eleven **(Text begins on page 44.)**

```
IMPLEMENTATION MODULE LongNumbers;
(* Routines to handle HEX digits for the X68000 cross assembler. *)
(* All but LongPut and LongWrite are limited to 8 digit numbers. *)

   FROM Files IMPORT
      FILE;

   IMPORT Files;   (* Write *)

   IMPORT Terminal;   (* Write *)

(*---
(* These objects are declared in the DEFINITION MODULE *)

   CONST
      DIGITS = 8;
      BASE = 16;

   TYPE
      LONG = ARRAY [1..DIGITS] OF INTEGER;
                                                         ---*)

   CONST
      Zero = 30H;
      Nine = 39H;
      hexA = 41H;
      hexF = 46H;


   PROCEDURE LongClear (VAR A : LONG);
   (* Sets A to Zero *)

      VAR
         i : CARDINAL;

      BEGIN
         FOR i := 1 TO DIGITS DO
            A[i] := 0;
         END;
      END LongClear;


   PROCEDURE LongAdd (A, B : LONG; VAR Result : LONG);
   (* Add two LONGs, giving Result *)

      VAR
         Carry : INTEGER;
         i : CARDINAL;

      BEGIN
         Carry := 0;
         FOR i := 1 TO DIGITS DO
            Result[i] := (A[i] + Carry) + B[i];
            IF Result[i] >= BASE THEN
               Result[i] := Result[i] - BASE;
               Carry := 1;
            ELSE
               Carry := 0;
            END;
         END;
      END LongAdd;


   PROCEDURE LongSub (A, B : LONG; VAR Result : LONG);
   (* Subtract two LONGs (A - B), giving Result *)

      VAR
         Borrow : INTEGER;
         i : CARDINAL;

      BEGIN
         Borrow := 0;
         FOR i := 1 TO DIGITS DO
            Result[i] := (A[i] - Borrow) - B[i];
            IF Result[i] < 0 THEN
               Result[i] := Result[i] + BASE;
               Borrow := 1;
            ELSE
               Borrow := 0;
            END;
         END;
      END LongSub;


   PROCEDURE CardToLong (n : CARDINAL; VAR A : LONG);
   (* Converts CARDINALs to LONGs *)

      VAR
         i : CARDINAL;

      BEGIN
         LongClear (A);


         i := 1
```

End Listing Eleven

Listing Twelve

```
IMPLEMENTATION MODULE CmdLin2;
(* Parses command line - returns pointer to an array of pointer to strings *)

   FROM SYSTEM IMPORT
      ADDRESS, ADR;


   CONST
      MAXARGS = 5;


   VAR
      CommandLine[80H] : ARRAY [0..7FH] OF CHAR;
      Arguments : ARRAY [0..MAXARGS - 1] OF ADDRESS;

   PROCEDURE ReadCmdLin (VAR ArgC : CARDINAL; VAR ArgV : ADDRESS);
   (* Gives count of items in command line, and an array of pointer to them *)

      VAR
         i, C : CARDINAL;

      BEGIN
         IF ORD (CommandLine[0]) = 0 THEN
            ArgC := 0;   (* Nothing in Command Tail Buffer *)
            ArgV := NIL;
         ELSE
            i := 1;   C := 0;

            LOOP
               WHILE CommandLine[i] = ' ' DO   (* Skip Blanks *)
                  INC (i);
               END;

               IF CommandLine[i] = 0C THEN   (* end of tail buffer *)
                  EXIT;
               ELSE
                  Arguments[C] := ADR (CommandLine[i]);
                  INC (C);
                  IF C = MAXARGS THEN
                     EXIT;
                  END;
               END;

               WHILE CommandLine[i] # ' ' DO   (* Advance to next Argument *)
                  INC (i);
                  IF CommandLine[i] = 0C THEN
                     EXIT;
                  END;
               END;

               CommandLine[i] := 0C;   (* Terminate Argument *)
               INC (i);
            END;   (* LOOP *)

            CommandLine[0] := 0C;   (* Command Tail must only be used once *)
            ArgC := C;
            ArgV := ADR (Arguments);
         END;
      END ReadCmdLin;

END CmdLin2.
```

End Listing Twelve

Listing Thirteen

```
IMPLEMENTATION MODULE Parser;
(* Reads the Source file, and splits each *)
(* line into Label, OpCode & Operand(s).  *)

   FROM Strings IMPORT
      STRING;

   FROM Files IMPORT
      FILE, EOF, Read;

   FROM ErrorX68 IMPORT
      ErrorType, Error;

   IMPORT ASCII;


(*---
(* These objects are declared in the DEFINITION MODULE *)
   CONST
      TokenSize = 8;
      OperandSize = 20;

   TYPE
      TOKEN = ARRAY [0..TokenSize] OF CHAR;
      OPERAND = ARRAY [0..OperandSize] OF CHAR;

   VAR
      OpLoc, SrcLoc, DestLoc : CARDINAL;   (* location of line parts *)
      Line : STRING;
      LineCount : CARDINAL;
                                                         ---*)

   PROCEDURE GetLine (f : FILE; VAR EndFile : BOOLEAN);
   (* Inputs a Line -- up to 80 characters ending in cr/lf -- from a file. *)

      CONST
         MAXLINE = 80;

      VAR
         i : CARDINAL;
         ch : CHAR;

      PROCEDURE Get (VAR c : CHAR) : CHAR;
         BEGIN
            IF NOT EOF (f) THEN
               Read (f, c);
               RETURN c;
            ELSE
               EndFile := TRUE;
            END;
         END Get;

      BEGIN   (* GetLine *)
         EndFile := FALSE;

         i := 0;
         WHILE (i < MAXLINE) AND (Get (ch) # ASCII.lf) AND (NOT EndFile) DO
            Line[i] := ch;
            INC (i);
         END;

         IF Line[i - 1] = ASCII.cr THEN   (* Strip cr/lf - terminate with 0C *)
            Line[i - 1] := 0C;
         ELSE
            Line[i] := 0C;
         END;

         INC (LineCount);
      END GetLine;


   PROCEDURE SplitLine (VAR Label, OpCode : TOKEN;
                        VAR SrcOp, DestOp : OPERAND);
   (* Separates TOKENs & OPERANDs from Line. *)
```

(continued on page 82)

Circle **no. 147** on reader service card.

68K ASSEMBLER

Listing Thirteen

(Listing continued, text begins on page 44.)

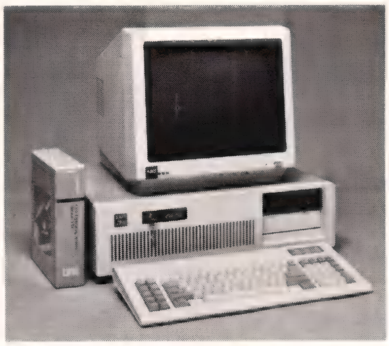
```
CONST
   Quote = 47C;
   StringMAX = 12;

VAR
   i, j : CARDINAL;
   ParCnt : INTEGER;   (* Tracks open parentheses *)
   c : CHAR;
   InQuotes : BOOLEAN;

PROCEDURE Cap (ch : CHAR) : CHAR;
   BEGIN
      IF InQuotes THEN
         RETURN (ch);
      ELSE
         RETURN CAP (ch);
      END;
   END Cap;

PROCEDURE White (ch : CHAR) : BOOLEAN;
   BEGIN
      RETURN ((ch = ASCII.ht) OR (ch = ' '));
   END White;

PROCEDURE Delimiter (ch : CHAR) : BOOLEAN;
   BEGIN
      RETURN ((NOT InQuotes) AND
             ((ch = ASCII.ht) OR (ch = ' ') OR (ch = 0C)));
   END Delimiter;

PROCEDURE OpDelimiter (ch : CHAR) : BOOLEAN;
   BEGIN
      RETURN ((NOT InQuotes) AND (ch = ',') AND (ParCnt = 0));
   END OpDelimiter;

PROCEDURE Done (ch : CHAR) : BOOLEAN;
(* look for start of comment or NULL terminator *)
   BEGIN
      RETURN ((ch = ';') OR (ch = 0C) OR ((ch = '*') AND (i = 0)));
   END Done;


BEGIN   (* SplitLine *)
   i := 0;
   InQuotes := FALSE;

   IF Done (Line[i]) THEN   (* look for blank or all-comment line *)
      RETURN;
   END;

   IF White (Line[i]) THEN
      INC (i);

   WHILE White (Line[i]) DO
      INC (i);   (* Skip spaces & tabs *)
   END;
ELSE   (* Found a Label *)
   j := 0;
   c := Line[i];
   WHILE (NOT Delimiter (c)) AND (j < TokenSize) DO
      Label[j] := CAP (c);
      INC (i);   INC (j);
      c := Line[i];
   END;
   Label[j] := 0C;   (* terminate Label string *)
   IF j = TokenSize THEN
      Error (i, TooLong);
   END;
   WHILE NOT Delimiter (Line[i]) DO
      INC (i);   (* Skip remainder of Too-Long Token *)
   END;
END;

WHILE White (Line[i]) DO
   INC (i);
END;

IF Done (Line[i]) THEN
   RETURN;
ELSE   (* Found an OpCode *)
   OpLoc := i;
   j := 0;
   c := Line[i];
   WHILE (NOT Delimiter (c)) AND (j < TokenSize) DO
      OpCode[j] := CAP (c);
      INC (i);   INC (j);
      c := Line[i];
   END;
   OpCode[j] := 0C;
   IF j = TokenSize THEN
      Error (i, TooLong);
   END;
   WHILE NOT Delimiter (Line[i]) DO
      INC (i);   (* Skip remainder of Too-Long Token *)
   END;
END;

WHILE White (Line[i]) DO
   INC (i);
END;

IF Done (Line[i]) THEN
   RETURN;
ELSE   (* Found 1st Operand *)
   SrcLoc := i;
   j := 0;
   ParCnt := 0;
   c := Line[i];
   IF c = Quote THEN   (* String Constant *)
      SrcOp[j] := c;
      INC (i);   INC (j);
      REPEAT
         c := Line[i];
         SrcOp[j] := c;
         INC (i);   INC (j);
      UNTIL (c = Quote) OR (j > StringMAX) OR (c = 0C);
      SrcOp[j] := 0C;
      IF j > StringMAX THEN
         Error (i, TooLong);
      END;
      RETURN;  (* second operand not allowed after string constant *)
   ELSE   (* Normal Operand *)
      WHILE (NOT Delimiter (c))
       AND (NOT OpDelimiter (c))
        AND (j < OperandSize) DO
         IF c = Quote THEN
```

(continued on page 84)

AMERICAN MICRO TECHNOLOGY
presents

STARRING

XT-PLUS AMT 286 and Introducing AT*jr*



XT-PLUS

IBM PC XT Compatible, 4.77MHz Clock, 640K Mother Board, 8088 Intel Chip, Keyboard, 135 watt Power Supply, Floppy Disk Controller, Printer Port, Serial Port, Game Port, Clock w/Battery Backup, Two Disk Drives . . . . . . **$699.**

Specials *

Floppy Disc Controller . . . . . . **$29.**
Monochrome Graphics Card/PP . . . . **79.**
Disk I/O Card FDC, PP/SP, Game, Battery Clock . . . . . . **79.**
1200 Baud Modem (½ size) . . **159**
XT-Mother Board "O"K expandable to 640K . . . . **99.**
Mother Board for AT . . . . . . **650.**
Keyboard AT . . . . **65.**
20MB Drive with Controller . . **425.**
Power Supply 135 watts . . . . **65.**
Power Supply 200 watts . . . **129.**
XT Chassis . . . . **31.**
AT Chassis . . . . **89.**
8MHz IBM PC XT Compatible Mother Board "O"K . . . . . . **175.**
30MB Drive with Controller . . **590.**
EGA Card . . . . **275.**

AMT 286

IBM PC AT Compatible Computer (STM Board) 6 MHz, 640K Memory, Keyboard, Clock & Battery on Board, Floppy & Hard Disk Controller, 1.2MB Floppy, 192 watts Power Supply **$1499.***

AMT 286-e

IBM PC AT Compatible Computer (Atronics Mother Board), 8 MHz Clock (enhanced version), 640KB Memory expandable to 1MB, AT layout Keyboard, Floppy & Hard Disk Controller by Western Digital, 192 watts Power Supply, 1.2MB Floppy, 20MB Hard Disk, Socket for 80287 . . . . **$1999.**

"O" Wait State Option . . . . **$300.**

PLEASE DON'T CALL TICKETRON . . .
FOR FAST SERVICE CALL US

(714) 972-2945
TWX 5106003265

AT*jr*

IBM PC Compatible, 2 to 5 times faster than IBM PC, Zero Wait State, Eight Slots, 135 watts Power Supply, 8 MHz & 4.77 MHz Clock (Dual Speed Hardware & Software switchable), 640KB on Mother Board, V-20 or 8088-2 Processor, AT layout Keyboard, Floppy Disk Controller, Two Drives 360KB each, Runs Lotus 123, Wordstar, dBase II & III, Flight Simulator and more **$699.***

*Prices for quantity purchases only
Registered trademark of IBM Corporation

AMERICAN MICRO TECHNOLOGY
1322 E. EDINGER SANTA ANA, CA 92705

Prices and Availability subject to change without notice.

Circle **no. 297** on reader service card.

Listing Thirteen **(Listing continued, text begins on page 44.)**

```
            InQuotes := NOT InQuotes;    (* Toggle Switch *)
         END;
         IF NOT InQuotes THEN
            IF c = '(' THEN
               INC (ParCnt);
            END;
            IF c = ')' THEN
               DEC (ParCnt);
            END;
         END;
         SrcOp[j] := Cap (c);    (* Switched CAP function *)
         INC (i);   INC (j);
         c := Line[i];
      END;
      SrcOp[j] := 0C;
      IF j = OperandSize THEN
         Error (i, TooLong);
      END;
   END;
   WHILE (NOT Delimiter (Line[i])) AND (NOT OpDelimiter (Line[i])) DO
      INC (i);   (* Skip remainder of Too-Long Operand *)
   END;
END;

IF NOT OpDelimiter (Line[i]) THEN
   RETURN;   (* because only one OPERAND *)
ELSE   (* Found 2nd Operand *)
   INC (i);   (* Skip OpDelimiter (comma) *)
   DestLoc := i;
   j := 0;
   c := Line[i];
   WHILE (NOT Delimiter (c)) AND (j < OperandSize) DO
      DestOp[j] := CAP (c);
      INC (i);   INC (j);
      c := Line[i];
   END;
   DestOp[j] := 0C;
   IF j = OperandSize THEN
      Error (i, TooLong);
   END;
END;
END SplitLine;


PROCEDURE LineParts (f : FILE; VAR EndFile : BOOLEAN;
                     VAR Label, OpCode : TOKEN;
                     VAR SrcOp, DestOp : OPERAND);
(* Reads line, breaks into tokens, on-passes to symbol & code generators *)

   BEGIN
      Line := "";
      GetLine (f, EndFile);   (* read a line from the file *)

      Label := "";   OpCode := "";   SrcOp := "";   DestOp := "";
      IF EndFile THEN
         Error (0, EndErr);
      ELSE
         SplitLine (Label, OpCode, SrcOp, DestOp);
      END;
   END LineParts;


BEGIN   (* MODULE Initialization *)
   OpLoc := 0;   SrcLoc := 0;   DestLoc := 0;   LineCount := 0;
END Parser.
```

End Listing Thirteen

Listing Fourteen

```
IMPLEMENTATION MODULE SymbolTable;
(* Initializes symbol table.  Maintains list of all labels, *)
(* along with their values.  Provides access to the list.   *)

   FROM LongNumbers IMPORT
      LONG, LongClear;

   FROM Parser IMPORT
      TOKEN;

   FROM Strings IMPORT
      CompareStr;


   CONST
      MAXSYM = 500;   (* Maximum entries in Symbol Table *)


   TYPE
      SYMBOL = RECORD
                  Name : TOKEN;
                  Value : LONG;
               END;

   VAR
      SymTab : ARRAY [1..MAXSYM] OF SYMBOL;
      Next : CARDINAL;   (* Array index into next entry in Symbol Table *)
      Top : INTEGER;   (* Last used array position as seen by Sort *)


   PROCEDURE FillSymTab (Label : TOKEN; Value : LONG; VAR Full : BOOLEAN);
   (* Add a symbol to the table *)
      BEGIN
         IF Next <= MAXSYM THEN
            SymTab[Next].Name := Label;
            SymTab[Next].Value := Value;
            INC (Next);
            Full := FALSE;
         ELSE
            Full := TRUE;
         END;
      END FillSymTab;


   PROCEDURE SortSymTab (VAR NumSyms : CARDINAL);
   (* Sort symbols into alphabetical order *)

      VAR
         i, j, gap : INTEGER;   (* Shell Sort causes j to go negative *)
         Temp : SYMBOL;

      PROCEDURE Swap;
         BEGIN
            Temp := SymTab[j];
            SymTab[j] := SymTab[j + gap];
            SymTab[j + gap] := Temp;
         END Swap;

      BEGIN   (* Sort *)
         Top := Next - 1;

         gap := (Top + 1) DIV 2;
         WHILE gap > 0 DO
            i := gap;
            WHILE i <= Top DO
               j := i - gap;
               WHILE j >= 1 DO
                  IF CompareStr (SymTab[j].Name, SymTab[j + gap].Name) > 0 THEN
                     Swap;
                  END;
                  j := j - gap;
               END;
               INC (i);
            END;
            gap := gap DIV 2;
         END;

         NumSyms := Top;
      END SortSymTab;


   PROCEDURE ReadSymTab (LABEL : ARRAY OF CHAR;
                         VAR Value : LONG; VAR Duplicate : BOOLEAN) : BOOLEAN;
   (* Passes Value of Label to calling program -- returns FALSE if the *)
   (* Label is not defined.  Also checks for Multiply Defined Symbols  *)

      CONST
         GoLower = -1;
         GoHigher = +1;

      VAR
         i, j, mid : INTEGER;
         Search : INTEGER;
         Found : BOOLEAN;
         c : CHAR;
         Label : TOKEN;

      BEGIN
         LongClear (Value);
         Duplicate := FALSE;

         i := 0;
         REPEAT
            c := LABEL[i];
            Label[i] := c;
            INC (i);
         UNTIL (c = 0C) OR (i > 8);

         IF c # 0C THEN   (* Operand label too long --> Undefined *)
            RETURN FALSE;
         END;

         i := 1;
         j := Top;
         Found := FALSE;

         REPEAT   (* Binary search *)
            mid := (i + j) DIV 2;
            Search := CompareStr (Label, SymTab[mid].Name);

            IF Search = GoLower THEN
               j := mid - 1;
            ELSIF Search = GoHigher THEN
               i := mid + 1;
            ELSE   (* Got It! *)
               Found := TRUE;
            END;
         UNTIL (j < i) OR Found;

         IF Found THEN
            IF mid > 1 THEN
               IF CompareStr (SymTab[mid].Name, SymTab[mid - 1].Name) = 0 THEN
                  Duplicate := TRUE;   (* Multiply Defined Symbol *)
               END;
            END;
            IF mid < Top THEN
               IF CompareStr (SymTab[mid].Name, SymTab[mid + 1].Name) = 0 THEN
                  Duplicate := TRUE;   (* Multiply Defined Symbol *)
               END;
            END;

            Value := SymTab[mid].Value;
            RETURN TRUE;
         ELSE
            RETURN FALSE;
         END;
      END ReadSymTab;


   PROCEDURE ListSymTab (i : CARDINAL; VAR Label : TOKEN; VAR Value : LONG);
   (* Returns the i-th item in the symbol table *)
      BEGIN
         IF i < Next THEN
            Label := SymTab[i].Name;
            Value := SymTab[i].Value;
         END;
      END ListSymTab;


BEGIN   (* MODULE Initialization *)
   FOR Next := 1 TO MAXSYM DO
      SymTab[Next].Name := "";
      LongClear (SymTab[Next].Value);
   END;

   Top := 0;
   Next := 1;
END SymbolTable.
```

End Listing Fourteen

(Listing Fifteen begins on page 86.)

Listing Fifteen **(Listing continued, text begins on page 44.)**

```
IMPLEMENTATION MODULE OperationCodes;
(* Initializes lookup table for Mnemonic OpCodes.  Searches the table *)
(* and returns the bit pattern along with address mode information.   *)

   FROM Files IMPORT
      FILE, FileState, Open, ReadRec, Close;

   FROM Terminal IMPORT
      WriteString, WriteLn;

   FROM Strings IMPORT
      STRING, CompareStr;

   FROM Parser IMPORT
      TOKEN;

   FROM ErrorX68 IMPORT
      ErrorType, Error;


   CONST
      FIRST = 1;        (* First 68000 OpCode *)
      LAST = 118;       (* Last 68000 OpCode *)


(*---
(* These objects are declared in the DEFINITION MODULE *)

   TYPE
      ModeTypeA = (RegMem3,        (* 0 = Register, 1 = Memory *)
                   Ry02,           (* Register Rx -- Bits 0-2 *)
                   Rx911,          (* Register Ry -- Bits 9-11 *)
                   Data911,        (* Immediate Data -- Bits 9-11 *)
                   CntR911,        (* Count Register or Immediate Data *)
                   Brnch,          (* Relative Branch *)
                   DecBr,          (* Decrement and Branch *)
                   Data03,         (* Used for VECT only *)
                   Data07,         (* MOVEQ *)
                   OpM68D,         (* Data *)
                   OpM68A,         (* Address *)
                   OpM68C,         (* Compare *)
                   OpM68X,         (* XOR *)
                   OpM68S,         (* Sign Extension *)
                   OpM68R,         (* Register/Memory *)
                   OpM37);         (* Exchange Registers *)

      ModeTypeB = (Bit811,         (* BIT operations - bits 8/11 as switch *)
                   Size67,         (* 00 = Byte, 01 = Word, 10 = Long *)
                   Size6,          (* 0 = Word, 1 = Long *)
                   Size1213A,      (* 01 = Byte, 11 = Word, 10 = Long *)
                   Size1213,       (* 11 = Word, 10 = Long *)
                   Exten,          (* OpCode extension required *)
                   EA05a,          (* Effective Address - ALL *)
                   EA05b,          (* Less 1 *)
                   EA05c,          (* Less 1, 11 *)
                   EA05d,          (* Less 9, 10, 11 *)
                   EA05e,          (* Less 1, 9, 10, 11 *)
                   EA05f,          (* Less 0, 1, 3, 4, 11 *)
                   EA05x,          (* Dual mode - OR/AND *)
                   EA05y,          (* Dual mode - ADD/SUB *)
                   EA05z,          (* Dual mode - MOVEM *)
                   EA611);         (* Used only by MOVE *)

      ModeA = SET OF ModeTypeA;
      ModeB = SET OF ModeTypeB;

                                                                   ---*)
   TYPE
      TableRecord = RECORD
                       Mnemonic : TOKEN;
                       Op : BITSET;
                       AddrModeA : ModeA;
                       AddrModeB : ModeB;
                    END;


   VAR
      Table68K : ARRAY [FIRST..LAST] OF TableRecord;
      i : CARDINAL;   (* index variable for initializing Table68K *)
      f : FILE;


   PROCEDURE Instructions (MnemonSym : TOKEN;
                           OpLoc : CARDINAL; VAR Op : BITSET;
                           VAR AddrModeA : ModeA; VAR AddrModeB : ModeB);
   (* Uses lookup table to find addressing mode & bit pattern of opcode. *)

      CONST
         GoLower = -1;
         GoHigher = +1;

      VAR
         Top, Bottom, Look : CARDINAL;   (* index to Op-code table *)
         Found : BOOLEAN;
         Search : INTEGER;

      BEGIN
         Bottom := FIRST;
         Top := LAST;
         Found := FALSE;

         REPEAT   (* Binary Search *)
            Look := (Bottom + Top) DIV 2;
            Search := CompareStr (MnemonSym, Table68K[Look].Mnemonic);

            IF Search = GoLower THEN
               Top := Look - 1;
            ELSIF Search = GoHigher THEN
               Bottom := Look + 1;
            ELSE   (* Got It! *)
               Found := TRUE;
            END;
         UNTIL (Top < Bottom) OR Found;

         IF Found THEN
            (* Return the instruction, mode, and address restristictions *)
            Op := Table68K[Look].Op;
            AddrModeA := Table68K[Look].AddrModeA;
            AddrModeB := Table68K[Look].AddrModeB;
         ELSE
            Error (OpLoc, NoCode);
         END;
      END Instructions;

BEGIN   (* MODULE Initialization *)
   IF Open (f, "OPCODE.DAT") # FileOK THEN
      WriteString ("Can't Find 'OPCODE.DAT'.");
      WriteLn;
      HALT;
   END;

   FOR i := FIRST TO LAST DO
      ReadRec (f, Table68K[i]);
   END;

   IF Close (f) # FileOK THEN
      (* Don't worry about it! *)
   END;
END OperationCodes.
```

End Listing Fifteen

Listing Sixteen

```
MODULE InitOperationCodes;
(* Module to construct the file containing the Operation Code Data Table *)

   FROM Files IMPORT
      FILE, FileState, Create, WriteRec, Close;

   FROM Terminal IMPORT
      WriteString, WriteLn;

   FROM Parser IMPORT
      TOKEN;


   CONST
      FIRST = 1;
      LAST = 118;

   TYPE
      ModeTypeA = (RegMem3,        (* 0 = Register, 1 = Memory *)
                   Ry02,           (* Register Rx -- Bits 0-2 *)
                   Rx911,          (* Register Ry -- Bits 9-11 *)
                   Data911,        (* Immediate Data -- Bits 9-11 *)
                   CntR911,        (* Count Register or Immediate Data *)
                   Brnch,          (* Relative Branch *)
                   DecBr,          (* Decrement and Branch *)
                   Data03,         (* Used for VECT only *)
                   Data07,         (* Branch & MOVEQ *)
                   OpM68D,         (* Data *)
                   OpM68A,         (* Address *)
                   OpM68C,         (* Compare *)
                   OpM68X,         (* XOR *)
                   OpM68S,         (* Sign Extension *)
                   OpM68R,         (* Register/Memory *)
                   OpM37);         (* Exchange Registers *)

      ModeTypeB = (Bit811,         (* BIT operations - bits 8/11 as switch *)
                   Size67,         (* 00 = Byte, 01 = Word, 10 = Long *)
                   Size6,          (* 0 = Word, 1 = Long *)
                   Size1213A,      (* 01 = Byte, 11 = Word, 10 = Long *)
                   Size1213,       (* 11 = Word, 10 = Long *)
                   Exten,          (* OpCode extension required *)
                   EA05a,          (* Effective Address - ALL *)
                   EA05b,          (* Less 1 *)
                   EA05c,          (* Less 1, 11 *)
                   EA05d,          (* Less 9, 10, 11 *)
                   EA05e,          (* Less 1, 9, 10, 11 *)
                   EA05f,          (* Less 0, 1, 3, 4, 11 *)
                   EA05x,          (* Dual mode - OR/AND *)
                   EA05y,          (* Dual mode - ADD/SUB *)
                   EA05z,          (* Dual mode - MOVEM *)
                   EA611);         (* Used only by MOVE *)

      ModeA = SET OF ModeTypeA;
      ModeB = SET OF ModeTypeB;

      TableRecord = RECORD
                       Mnemonic : TOKEN;
                       Op : BITSET;
                       AddrModeA : ModeA;
                       AddrModeB : ModeB;
                    END;


   VAR
      Table68K : ARRAY [FIRST..LAST] OF TableRecord;
      i : CARDINAL;   (* index variable for initializing Table68K *)
      f : FILE;   (* "OPCODE.DAT" *)


BEGIN
   i := 1;
   WITH Table68K[i] DO
      Mnemonic := "ABCD";
      Op := {15, 14, 8};
      AddrModeA := ModeA{Rx911, RegMem3, Ry02};
      AddrModeB := ModeB{};
   END;

   INC (i);
   WITH Table68K[i] DO
      Mnemonic := "ADD";
      Op := {15, 14, 12};
      AddrModeA := ModeA{OpM68D};
      AddrModeB := ModeB{EA05y};
   END;

   INC (i);
   WITH Table68K[i] DO
      Mnemonic := "ADDA";
      Op := {15, 14, 12};
      AddrModeA := ModeA{OpM68A};
      AddrModeB := ModeB{EA05a};
   END;


   INC (i);
   WITH Table68K[i] DO
      Mnemonic := "ADDI";
      Op := {10, 9};
      AddrModeA := ModeA{};
      AddrModeB := ModeB{Size67, EA05e, Exten};
   END;
```

(continued on page 88)

Listing Sixteen **(Listing continued, text begins on page 44.)**

```
      INC (i);
      WITH Table68K[i] DO
         Mnemonic := "ADDQ";
         Op := {14, 12};
         AddrModeA := ModeA{Data911};
         AddrModeB := ModeB{Size67, EA05d};
      END;

      INC (i);
      WITH Table68K[i] DO
         Mnemonic := "ADDX";
         Op := {15, 14, 12, 8};
         AddrModeA := ModeA{RegMem3, Rx911, Ry02};
         AddrModeB := ModeB{Size67};
      END;

      INC (i);
      WITH Table68K[i] DO
         Mnemonic := "AND";
         Op := {15, 14};
         AddrModeA := ModeA{OpM68D};
         AddrModeB := ModeB{EA05x};
      END;

      INC (i);
      WITH Table68K[i] DO
   Mnemonic := "ANDI";
   Op := {9};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e, Size67, Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ASL";
   Op := {15, 14, 13, 8};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ASR";
   Op := {15, 14, 13};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BCC";
   Op := {14, 13, 10};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BCHG";
   Op := {6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e, Exten, Bit811};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BCLR";
   Op := {7};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e, Exten, Bit811};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BCS";
   Op := {14, 13, 10, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BEQ";
   Op := {14, 13, 10, 9, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BGE";
   Op := {14, 13, 11, 10};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BGT";
   Op := {14, 13, 11, 10, 9};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BHI";
   Op := {14, 13, 9};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BLE";
   Op := {14, 13, 11, 10, 9, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BLS";
   Op := {14, 13, 9, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BLT";
   Op := {14, 13, 11, 10, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BMI";
   Op := {14, 13, 11, 9, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BNE";
   Op := {14, 13, 10, 9};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BPL";
   Op := {14, 13, 11, 9};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BRA";
   Op := {14, 13};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BSET";
   Op := {7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e, Exten, Bit811};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BSR";
   Op := {14, 13, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BTST";
   Op := {};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05c, Exten, Bit811};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BVC";
   Op := {14, 13, 11};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "BVS";
   Op := {14, 13, 11, 8};
   AddrModeA := ModeA{Brnch};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "CHK";
   Op := {14, 8, 7};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{EA05b};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "CLR";
   Op := {14, 9};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "CMP";
   Op := {15, 13, 12};
   AddrModeA := ModeA{OpM68C};
   AddrModeB := ModeB{EA05a};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "CMPA";
   Op := {15, 13, 12};
   AddrModeA := ModeA{OpM68A};
   AddrModeB := ModeB{EA05a};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "CMPI";
   Op := {11, 10};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e, Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "CMPM";
   Op := {15, 13, 12, 8, 3};
   AddrModeA := ModeA{Rx911, Ry02};
   AddrModeB := ModeB{Size67};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBCC";
   Op := {14, 12, 10, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBCS";
   Op := {14, 12, 10, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBEQ";
   Op := {14, 12, 10, 9, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBF";
   Op := {14, 12, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBGE";
   Op := {14, 12, 11, 10, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBGT";
   Op := {14, 12, 11, 10, 9, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBHI";
   Op := {14, 12, 9, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBLE";
   Op := {14, 12, 11, 10, 9, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBLS";
   Op := {14, 12, 9, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBLT";
   Op := {14, 12, 11, 10, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBMI";
   Op := {14, 12, 11, 9, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBNE";
   Op := {14, 12, 10, 9, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBPL";
   Op := {14, 12, 11, 9, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBRA";
   Op := {14, 12, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBT";
   Op := {14, 12, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBVC";
   Op := {14, 12, 11, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DBVS";
```

(continued on page 90)

Listing Sixteen (Listing continued, text begins on page 44.)

```
   Op := {14, 12, 11, 8, 7, 6, 3};
   AddrModeA := ModeA{DecBr};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DIVS";
   Op := {15, 8, 7, 6};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{EA05b};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "DIVU";
   Op := {15, 7, 6};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{EA05b};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "EOR";
   Op := {15, 13, 12};
   AddrModeA := ModeA{OpM68X};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "EORI";
   Op := {11, 9};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e, Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "EXG";
   Op := {15, 14, 8};
   AddrModeA := ModeA{OpM37};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "EXT";
   Op := {14, 11};
   AddrModeA := ModeA{OpM68S};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ILLEGAL";
   Op := {14, 11, 9, 7, 6, 5, 4, 3, 2};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "JMP";
   Op := {14, 11, 10, 9, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05f};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "JSR";
   Op := {14, 11, 10, 9, 7};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05f};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "LEA";
   Op := {14, 8, 7, 6};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{EA05f};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "LINK";
   Op := {14, 11, 10, 9, 6, 4};
   AddrModeA := ModeA{Ry02};
   AddrModeB := ModeB{Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "LSL";
   Op := {15, 14, 13, 9, 8, 3};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "LSR";
   Op := {15, 14, 13, 9, 3};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MOVE";
   Op := {};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size1213A, EA611};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MOVEA";
   Op := {6};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{Size1213, EA05a};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MOVEM";
   Op := {14, 11, 7};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size6, EA05z, Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MOVEP";
   Op := {3};
   AddrModeA := ModeA{OpM68R};
   AddrModeB := ModeB{Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MOVEQ";
   Op := {14, 13, 12};
   AddrModeA := ModeA{Data07};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MULS";
   Op := {15, 14, 8, 7, 6};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{EA05b};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "MULU";
   Op := {15, 14, 7, 6};
   AddrModeA := ModeA{Rx911};
   AddrModeB := ModeB{EA05b};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "NBCD";
   Op := {14, 11};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "NEG";
   Op := {14, 10};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "NEGX";
   Op := {14};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "NOP";
   Op := {14, 11, 10, 9, 6, 5, 4, 0};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "NOT";
   Op := {14, 10, 9};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "OR";
   Op := {15};
   AddrModeA := ModeA{OpM68D};
   AddrModeB := ModeB{EA05x};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ORI";
   Op := {};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e, Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "PEA";
   Op := {14, 11, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05f};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "RESET";
   Op := {14, 11, 10, 9, 6, 5, 4};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ROL";
   Op := {15, 14, 13, 10, 9, 8, 4, 3};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ROR";
   Op := {15, 14, 13, 10, 9, 4, 3};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ROXL";
   Op := {15, 14, 13, 10, 8, 4};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ROXR";
   Op := {15, 14, 13, 10, 4};
   AddrModeA := ModeA{CntR911};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "RTE";
   Op := {14, 11, 10, 9, 6, 5, 4, 1, 0};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "RTR";
   Op := {14, 11, 10, 9, 6, 5, 4, 2, 1, 0};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "RTS";
   Op := {14, 11, 10, 9, 6, 5, 4, 2, 0};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SBCD";
   Op := {15, 8};
   AddrModeA := ModeA{Rx911, RegMem3, Ry02};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SCC";
   Op := {14, 12, 10, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SCS";
   Op := {14, 12, 10, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SEQ";
   Op := {14, 12, 10, 9, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SF";
   Op := {14, 12, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SGE";
   Op := {14, 12, 11, 10, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SGT";
   Op := {14, 12, 11, 10, 9, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SHI";
   Op := {14, 12, 9, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SLE";
   Op := {14, 12, 11, 10, 9, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SLS";
   Op := {14, 12, 9, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SLT";
   Op := {14, 12, 11, 10, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SMI";
   Op := {14, 12, 11, 9, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;
```

(continued on page 92)

Listing Sixteen (Listing continued, text begins on page 44.)

```
INC (i);
WITH Table68K[i] DO
   Mnemonic := "SNE";
   Op := {14, 12, 10, 9, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SPL";
   Op := {14, 12, 11, 9, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "ST";
   Op := {14, 12, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "STOP";
   Op := {14, 11, 10, 9, 6, 5, 4, 1};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SUB";
   Op := {15, 12};
   AddrModeA := ModeA{OpM68D};
   AddrModeB := ModeB{EA05y};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SUBA";
   Op := {15, 12};
   AddrModeA := ModeA{OpM68A};
   AddrModeB := ModeB{EA05a};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SUBI";
   Op := {10};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e, Exten};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SUBQ";
   Op := {14, 12, 8};
   AddrModeA := ModeA{Data911};
   AddrModeB := ModeB{Size67, EA05d};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SUBX";
   Op := {15, 12, 8};
   AddrModeA := ModeA{Rx911, RegMem3, Ry02};
   AddrModeB := ModeB{Size67};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SVC";
   Op := {14, 12, 11, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SVS";
   Op := {14, 12, 11, 8, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "SWAP";
   Op := {14, 11, 6};
   AddrModeA := ModeA{Ry02};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "TAS";
   Op := {14, 11, 9, 7, 6};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "TRAP";
   Op := {14, 11, 10, 9, 6};
   AddrModeA := ModeA{Data03};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "TRAPV";
   Op := {14, 11, 10, 9, 6, 5, 4, 2, 1};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "TST";
   Op := {14, 11, 9};
   AddrModeA := ModeA{};
   AddrModeB := ModeB{Size67, EA05e};
END;

INC (i);
WITH Table68K[i] DO
   Mnemonic := "UNLK";
   Op := {14, 11, 10, 9, 6, 4, 3};
   AddrModeA := ModeA{Ry02};
   AddrModeB := ModeB{};

END;

   IF Create (f, "OPCODE.DAT") # FileOK THEN
      WriteString ("Unable to create OpCode File.");
      WriteLn;
      HALT;
   END;

   FOR i := FIRST TO LAST DO
      WriteRec (f, Table68K[i]);
   END;

   IF Close (f) # FileOK THEN
      WriteString ("Unable to close OpCode File.");
      WriteLn;
   END;
END InitOperationCodes.
```

End Listing Sixteen

(to be continued next month)

CRYPTOGRAPHER'S TOOLBOX

Listing One (Text begins on page 58.)

```
/*
** cypher.c          File Cypher Program  by F.A.Scacchitti  9/11/85
**
**                   Written in Small-C Version 2.10 or later
**
**                   Copies from original file to encrypted file
**                   using cypher key(s) passed to  encode or decode.
*/

#include <stdio.h>

#define BUFSIZE 16384

int fdin, fdout;     /* file  i/o channel pointers */
int n, count;
char *inbuf, *key;

main(argc,argv) int argc, argv[]; {

   inbuf = malloc(BUFSIZE);

/*
** Open file streams
*/
   if(argc < 4) {
      printf("\ncypher usage: cypher <source file> <new file>
 <key1> <key2> . . . <keyN> <CR>\n");
      exit();
   }
   if((fdin = fopen(argv[1],"r")) == NULL) {
      printf("\nUnable to open %s\n", argv[1]);
      exit();
   }
   if((fdout = fopen(argv[2],"w")) == NULL) {
      printf("\nUnable to create %s\n", argv[2]);
      exit();
   }

/*
** Read file - encode it - Write new file
*/
   do {
      printf("-reading file\n");

      count = read(fdin,inbuf,BUFSIZE);

      n=3;

      while(n++ <argc){
         key = argv[n-1];
         cypher(inbuf,count,key);
      }
      printf("-writing %d byte file\n\n", count);

      write(fdout,inbuf,count);

   } while(count == BUFSIZE);

   /* close up shop */

      fclose(fdin);
      fclose(fdout);
}
```

End Listing One

Listing Two

```
/*        cypher1.c  Cypher module                     by F.A.Scacchitti
**                                                     10/10/85
**
**        Simple cypher module - encodes directly with user keys
**
*/

#include <stdio.h>

static int i, n, keylength;

cypher1(buffer, num, code) char *buffer, *code; int num;{

/*
** get keylength for each key
*/

   keylength = 0;
   while(code[keylength++] != NULL);
   keylength--;

/*
** encrypt the file with each key
*/

   printf("-encoding/decoding buffer\n");

   for(i=0; i<=num; i++)
      buffer[i] = buffer[i] ^ code[i % keylength];
}
```

End Listing Two

Listing Three

```
/*        cypher2.c Cypher module                    by F.A.Scacchitti
**                                                   10/11/85
**
**      Complex cypher module - generates a key of some prime length
**                              between 1024 and 2028 bytes then
**                              encrypts the buffer with this key
**
*/

#include <stdio.h>

#define NEWBUF 2000
#define NUMPRIMES 50

static int i, n, index, length, sum, keylength;
static char *newkey;
static int prime[] = {1009, 1999, 1013, 1997, 1019,
                      1993, 1021, 1987, 1031, 1979,
                      1033, 1973, 1039, 1951, 1049,
                      1949, 1051, 1933, 1061, 1931,
                      1063, 1913, 1069, 1907, 1087,
                      1901, 1091, 1889, 1093, 1879,
                      1097, 1877, 1103, 1873, 1109,
                      1871, 1117, 1867, 1123, 1861,
                      1129, 1847, 1151, 1831, 1153,
                      1823, 1163, 1813, 1171, 1803};

cypher1(buffer, num, code) char *buffer, *code; int num;{

/*
** allocate a buffer for the generated key
*/
   newkey = malloc(NEWBUF);

/*
** get keylength and sumcheck for each key
*/

   keylength = sum = 0;
   while((n = code[keylength]) != NULL){
      sum += n;
      keylength++;
   }

/*
** Select a prime and generate a new key that length
*/

   length = prime[sum % NUMPRIMES];

   printf("-generating a %d byte key\n",length);

   for(i=0; i<length; i++){
      index = i % keylength;
      sum = code[index] + sum & 255;
      newkey[i] = code[index] ^ sum;
   }

/*
** encrypt the file with the generated key
*/

   printf("-encoding/decoding buffer\n");

   for(i=0; i<=num; i++)
      buffer[i] = buffer[i] ^ newkey[i % length];

/*
** get rid of the buffer
*/

   cfree(newkey);

}
```

End Listing Three

Listing Four

```
/*        cypher3.c Cypher module                    by F.A.Scacchitti
**                                                   11/09/85
**
**      Complex cypher module - generates a key of some prime length
**                              between 1024 and 2028 bytes then
**                              encrypts the buffer with this key
**                                                          or
**                              if key starts with a '-' (dash)
**                              calculate a transposition block size
**                              and invert (transpose) the file in
**                              this size blocks
**
*/

#include <stdio.h>

#define DASH 45
#define NEWBUF 2000
#define NUMPRIMES 50
```

(continued on next page)

CRYPTOGRAPHER'S TOOLBOX

Listing Four

(Listing continued, text begins on page 58.)

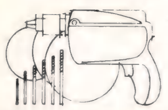
```
static int i, j, n, index, length, sum, keylength;
static char *tbuff, c;
static int prime[] = {1009, 1999, 1013, 1997, 1019,
                      1993, 1021, 1987, 1031, 1979,
                      1033, 1973, 1039, 1951, 1049,
                      1949, 1051, 1933, 1061, 1931,
                      1063, 1913, 1069, 1907, 1087,
                      1901, 1091, 1889, 1093, 1879,
                      1097, 1877, 1103, 1873, 1109,
                      1871, 1117, 1867, 1123, 1861,
                      1129, 1847, 1151, 1831, 1153,
                      1823, 1163, 1813, 1171, 1803};

cypher(buffer, num, code) char *buffer, *code; int num;{

/*
** allocate a buffer for the new key or transposition
*/
   tbuff = malloc(NEWBUF);

/*
** get keylength and sumcheck for each key
*/

   keylength = sum = 0;
   while((n = code[keylength]) != NULL){
      sum += n;
      keylength++;
   }

/*
** do we transpose or encode ?
*/

   if((c = *code) == DASH)
      transpose(buffer, num, code);
   else
      encode(buffer, num, code);

/*
** get rid of the buffer
*/

   cfree(tbuff);

}

/*
** Here's where we transpose
*/

transpose(buffer, num, code) char *buffer, *code; int num;{

   length = (((sum + keylength) % 16) & 15) + 2;

   printf("-transposing file by %d\n",length);

   index = 0;

   do{
      for(i = 0; i < length; i++){
         j = length - i - 1;
         tbuff[j] = buffer[index + i];
      }
      for(i = 0; i < length; i++){
         buffer[index + i] = tbuff[i];
      }
      index += length;

   }while(index < count);

}

/*
** Here's where we encode
*/

encode(buffer, num, code) char *buffer, *code; int num;{

/*
** Select a prime and generate a new key that length
*/

   length = prime[sum % NUMPRIMES];

   printf("-generating a %d byte key\n",length);

   for(i=0; i<length; i++){
      index = i % keylength;
      sum = code[index] + sum & 255;
      tbuff[i] = code[index] ^ sum;
   }

/*
** encrypt the file with the generated key
*/

   printf("-encoding/decoding buffer\n");

   for(i=0; i<=num; i++)
      buffer[i] = buffer[i] ^ tbuff[i % length];

}
```

End Listing Four

Listing Five

```
/*
** fv.c    File View/Compare Program    by F.A.Scacchitti  9/11/85
**
**                Written in Small-C Version 2.10 or later
**
**                Dumps contents of single file to screen
**                                     or
**                Dumps contents of 2 files and xored difference
**
**                Displays in hex and ascii form
*/

#include <stdio.h>

#define BUFSIZE 1024

int fdin1, fdin2;     /* file  i/o channel pointers */
int i, j, k,  val, count, total, offset, numdisp;
char *inbuf1, *inbuf2, *difbuf, c;

main(argc,argv) int argc, argv[]; {

   switch (argc) {
   case 2:
   case 3:
      inbuf1 = malloc(BUFSIZE);
      if((fdin1 = fopen(argv[1],"r")) == NULL) {
         printf("\nUnable to open %s\n", argv[1]);
         exit();
      }
      numdisp = 1;
      if(argc == 2) break;
   case 3:
      inbuf2 = malloc(BUFSIZE);
      difbuf = malloc(BUFSIZE);
      if((fdin2 = fopen(argv[2],"r")) == NULL) {
         printf("\nUnable to open %s\n", argv[2]);
         exit();
      }
      numdisp = 3;
      break;
   default:
      printf("\nfv usage: fv <file1>          -  dump file\n");
      printf("          fv <file1> <file2>  -  compare 2 files\n");
      exit();
   }
   total = offset = 0;

   printf("\n");

   do {
      count = read(fdin1,inbuf1,BUFSIZE);
      if(argc >= 3){
         read(fdin2,inbuf2,BUFSIZE);

         for(i=0; i< count; i++)
            difbuf[i] = inbuf1[i] ^ inbuf2[i];
      }

      for(i=0; i< count; i+=16){

         for(k=1; k <= numdisp; k++){

            switch (k) {
            case 2:
               offset = BUFSIZE;
               break;
            case 3:
               offset = 2 * BUFSIZE;
               break;
            default:
               offset = 0;
               break;
            }
            if(k < 3)
               printf("f-%d", k);
            else
               printf("dif");

            printf(" %04x  ",i+total);

            for(j=0; j<=15; j++){
               val = inbuf1[i + j + offset];
               printf("%02x ",val < 0 ? val - 65280 : val);

               if((c = bdos(6,255)) == 19){      /* hold on ^S */
                  if((c = getchx()) == 3)
                     exit();                      /* exit on ^C */

               }                                  /* continue on ^Q */

            }
            printf("  ");

            for(j=0; j<=15; j++){
               c = inbuf1[i + j + offset];
               if(c > 31)
                  putchar(c);
               else
                  if(c==0)
                     putchar(61);
                  else
                     putchar(94);
            }
            printf("\n");
            if(k == 3) printf("\n");
         }
      }
      total += count;

   } while(count == BUFSIZE);
```

(continued on next page)



CRYPTOGRAPHER'S TOOLBOX

Listing Five

(Listing continued, text begins on page 58.)

```
     /* close up shop */

   fclose(fdin1);
   if(argc == 3)
      fclose(fdin2);
}
```

End Listing Five

Listing Six

```
/*
** fstat.c    File Statistics Program    by F.A.Scacchitti 10/8/85
**
**                  Written in Small-C Version 2.10 or later
**
**                  Scans file and displays distribution of
**                  characters.
**                  Calculates and displays mean, mode, median
**                  and range of file.
**                  Displays histogram of distribution.
*/

#include <stdio.h>

#define BUFSIZE 16384

int fdin; /* file  pointer */
int i, j, temp, value, count, total, *file, *sorted;
int sum, hisum, meansum, himeansum, mean, eflag, changing;
int median, oddmedian, range, min, max, mode;
int *data, scale;
char c, *inbuf;

main(argc,argv) int argc, argv[]; {

   if(argc < 2) {
      printf("\nfstat usage: fstat <input file>\n");
      exit();
   }
   if((fdin = fopen(argv[1],"r")) == NULL) {
      printf("\nUnable to open file %s\n",argv[1]);
      exit();
   }

   inbuf = calloc(BUFSIZE,1);
   file = calloc(256,2);
   sorted = calloc(256,2);
   data = calloc(17,2);
   eflag = FALSE;
   sum = hisum = meansum = himeansum = mean = mode = j = 0;

   printf("reading the file-");

   do {
      count = read(fdin,inbuf,BUFSIZE);

      for(i=0; i< count; i++){

         value = inbuf[i];


         if(value < 0)
            value = 256 + value;
         file[value]++;
         if(++sum == 10000){
            hisum++;
            sum =0;
         }
         if((meansum += value) >= 10000){
            himeansum++;
            meansum -= 10000;
         }
      }
   } while(count == BUFSIZE);

/*
** Calculate the mean
*/

   printf("calculating mean-");

   do{
      if((meansum -= sum) < 0)
         if(himeansum > 0){
            himeansum--;
            meansum += 10000;
         }else{
            meansum += sum;
            eflag = TRUE;
            mean--;
         }
      if((himeansum -= hisum) < 0){
         himeansum += hisum;
         eflag = TRUE;
      }else{
         mean++;
      }
   }while(eflag == FALSE);

/*
```

```
** Calculate range, find mode min and max, fill the sorted array
*/

   printf("calculating range-");

   min = max = file[0];

   for(i = 0; i <= 255; i++){
      sorted[i] = file[i];
      if(file[i] > max){
         max = file[i];
         mode = i;
      }
      if(file[i] < min)

         min = file[i];
   }
   range = max - min + 1;

/*
** Sort the sorted array to calculate median
*/

   printf("sorting the array");

   changing = TRUE;

   while(changing){
      changing = FALSE;
      for(i = 0; i <= 254; i++)
         if(sorted[i] > sorted[i+1]){
            temp = sorted[i];
            sorted[i] = sorted[i+1];
            sorted[i+1] = temp;
            changing = TRUE;
         }
   }

   median = (sorted[128] + sorted[127]) / 2;
   oddmedian = (sorted[128] + sorted[127]) % 2;

/*
** Display the results
*/

   printf("\n    0    1    2    3    4    5    6    7
                8    9    A    B    C    D    E    F\n");

   for(i = 0; i <= 255; i++) {
      printf("%5d", file[i]);
      chkkbd();
   }

   printf("\n %d%04d characters read from file %s\n",
                                  hisum, sum, argv[1]);
   printf("file mean = %d  ",mean);
   if((himeansum || meansum) > 0)
      printf("%d%04d/%d%04d",himeansum,meansum,hisum,sum);
   printf("           mode = %d   ( %x hex)", mode, mode);
   printf("\n");
   printf("file median = %d", median);
   if(oddmedian)
      printf(" 1/2 ");
   else
      printf("      ");
   printf("     file range = %d  [ min = %d     max = %d ]\n",
                                  range, min, max,);
   printf("\nDepress spacebar to display histogram ");
   getchar();

/*
** Sum the data in 16 groups of 16 elements and find max. value
*/
   max = 0;
   for(i = 1; i <= 16; i++){
      for(j = 0; j <= 15; j++)
         data[i] += file[(i - 1) * 16 + j];
      if(data[i] > max)
         max = data[i];
   }
/*
** Calculate scaling for plot
*/
   scale = max / 50;
   temp = max % 50;
   if(temp / scale > 7)
      scale++;
   printf("   scale = %4d\n\n", scale);

/*
** Print data and plot of histogram
*/

   for(i = 0; i <= 15; i++){
      printf(" %3d to %3d = %5d ||",i  * 16, (i * 16) + 15,
                                            data[i + 1]);
      temp = data[i + 1] / scale;
      if(data[i + 1] % scale > 0)
         temp++;
      while(temp-- > 0)
         printf("*");
      printf("\n");
   }

/*
** close up shop
*/
   fclose(fdin);
}
```

(continued on next page)

CRYPTOGRAPHER'S TOOLBOX

Listing Six

(Listing continued, text begins on page 58.)

```
chkkbd(){ char c;

   if((c = bdos(6,255)) == 19)      /* hold on ^S */
      if((c = getchx()) == 3)
         exit();                     /* exit on ^C */
}                                    /* continue   */
```

End Listing Six

Listing Seven

```
/*
** makef.c    File Generator Program     by F.A.Scacchitti 10/7/85
**
**                 Written in Small-C Version 2.10 or later
**
**                 Creates a seqential file of characters from
**                 0 to 255 in blocks of 255 bytes.
**                 The sequential characters can be replaced
**                 with any single character desired.
*/

#include <stdio.h>

#define BUFSIZE 256


int fdout;           /* file  i/o channel pointers */
int i, n, num;
char *outbuf, value;

main(argc,argv) int argc, argv[]; {

/*
** Allocate memory for buffer
*/

   outbuf = malloc(BUFSIZE);

/*
** Check arguments passed and open file stream
*/
   if(argc < 3) {
      printf("\nmakef usage: makef <new file> <nnnn> [ddd]\n");
      printf(" nnnn - file size in 256 byte blocks\n");
      printf(" ddd  - optional alternate value in decimal\n");
      exit();
   }
   if((fdout = fopen(argv[1],"w")) == NULL) {
      printf("\nUnable to create %s\n", argv[1]);
      exit();
     }
/*
** Convert file size argument to integer
*/
   if((n = atoi(argv[2])) == NULL) exit();

/*
** Fill the buffer with 0 to 255 sequence
*/
   for(i = 0; i <=255; i++)
      outbuf[i] = i;

/*
** Refill the buffer with a single character if directed by argument
*/
   if(argc == 4)
      if((value = atoi(argv[3])) < 256)
         for(i = 0; i <=255; i++)
            outbuf[i] = value;

/*
** Write blocks to file
*/
   for(i=1; i <= n; i++)
      if((num = write(fdout,outbuf,BUFSIZE)) < BUFSIZE) exit();

/*
** Close up shop
*/

   fclose(fdout);
}
```

End Listing Seven

Listing Eight

```
/*
** sp.c          Search Pattern Program  by F.A.Scacchitti 10/15/85
**
**                 Written in Small-C Version 2.10 or later
**
**                 Searches file for repetitive pattern.
**
**
```

```c
*/
#include <stdio.h>

#define BUFSIZE 16384

int  fdin;           /* file  i/o channel pointers */
int i, j, n, block, start, depth, count, limit;
char *c, *inbuf;

main(argc,argv) int argc, argv[]; {

/*
** Set defaults
*/
   block = 128;
   depth = 4;
   start = 0;

/*
** Allocate memory for buffer
*/
   inbuf = malloc(BUFSIZE);

/*
** Check arguments passed and open file stream
*/
   if(argc < 2) {
      printf("\nsp usage: sp <source file> [nnnn] [dddd] [ssss] [x]\n");
      printf("    nnnn = block size to search    default = 128\n");
      printf("    dddd = minimum depth of comparison  default = 4\n");
      printf("    ssss = starting point in buffer default = 0\n");
      printf("    x - any char. gen's difference buffer ((n+1)-n)\n");
      exit();
   }
   if((fdin = fopen(argv[1],"r")) == NULL) {
      printf("\nUnable to open %s\n", argv[1]);
      exit();
   }

/*
** Convert optional inputs to integer and implement
*/
   if(argc > 2)
      if((n = atoi(argv[2])) != NULL)
         block = n;

   if(argc > 3)
      if((n = atoi(argv[3])) != NULL)
         depth = n;

   if(argc > 4)
      if((n = atoi(argv[4])) != NULL)
         start = n;

/*
** Fill the buffer with as much as possible
*/
   count = read(fdin,inbuf,BUFSIZE);

   limit = count - depth;

/*
** If there's a sixth argument, convert the file to numerical sequence
*/

   if(argc > 5){
      for(i = 0; i < count - 1; i++)
         inbuf[i] = inbuf[i + 1] - inbuf[i];
   }

   for(i = start; i < block; i++){
      printf("%c", (i % 10 == 0) ? '|' : '.');
      chkkbd();
      for(j = i + 1; j <= limit; j++)
         if(inbuf[i] == inbuf[j]){
            if((n = chkdepth(i, j, 0)) >= depth)
               printf("\nmatch at %d (%x hex) and %d (%x hex) -
                                  %d deep\n", i, i, j, j, n);
            if(n >= block) exit();
         }
   }
/*
** Close up shop
*/
   printf("\n");     /* Flush print buffer */

   fclose(fdin);
}

chkdepth(pointer, offset, k)int pointer, offset, k;{


   while(inbuf[pointer] == inbuf[offset] && k < count){
      pointer++;
      offset++;
      k++;
   }
   return(k);
}

chkkbd(){

char c;

   if((c = bdos(6,255)) == 19){        /* hold on ^S */
      if((c = getchx()) == 3)
         exit();                       /* exit on ^C */

   }                                   /* continue   */
}
```

End Listings

Listing One (Text begins on page 106.)

```
                name    window
                page    ,132

                title   BIOS Window Extension V1.1

comment         \

                Copyright 1984 John J. Seal

                The Graphic Software Company
                348 East Pratt Street
                Franklin, IN 46131

                This program may be used for any non-commercial purpose
                provided that this copyright notice is included.
                Commercial use is forbidden without the express written
                consent of The Graphic Software Company.

                This program allows a window to be defined on the display.
                All programs which use the BIOS Write TTY call for output
                will work within the window. Other BIOS calls may still be
                used for I/O to arbitrary screen positions.

                The window is defined by a pair of coordinates specified on
                the command line. The only absolute format requirements are
                that each coordinate pair start with a left parenthesis and
                be separated with a comma. Everything else is optional. The
                suggested command format is:

                window (ur,lc) to (lr,rc)

                ur = upper row
                lc = left column
                lr = lower row
                rc = right column

                The new functions serviced by the video interrupt (int 10)
                and their corresponding function codes are:

                ah = 14 Write TTY
                ah = 16 Set window coordinates
                ah = 17 Get window coordinates
                ah = 18 Get blanking attribute

                \

code            segment
                org     100h                    ;For COM conversion

                assume  cs:code

DOS_entry       label   far                     ;DOS entry point
                jmp     install

upper_left      label   word                    ;Window coordinates
left            db      0
upper           db      0
lower_right     label   word
right           db      79
lower           db      24

old_int         dd      ?                       ;Old interrupt vector

comment         \

                The new interrupt procedure first filters out the new
                services from the old, and passes all old service calls
                back to the BIOS.

                \

interrupt       proc    far
                sti
                cmp     ah,14                   ;Write TTY
                je      write_tty
                cmp     ah,16                   ;Set window
                je      set_window
                cmp     ah,17                   ;Get window
                je      get_window
                cmp     ah,18                   ;Get blanking
                je      get_blanking
bios:           jmp     old_int

comment         \

                Set window coordinates.

                This function call sets the coordinates of a window in the
                display area. The window is defined by specifying the upper
                left and lower right corners in terms of screen coordinates.
                The upper left corner of the screen is position (0,0). The
                specified corners are included in the window area.
                If the specified coordinates are legal then the window is
                cleared, the cursor is homed to the upper left corner, and
                al = 0. Otherwise, no action is taken and al = 1.
```

```
                Entry:  ah = 16 (function code)
                        cx = upper left corner
                        dx = lower right corner
                Exit:   al = fail flag (see above)

                All registers preserved except ax.

                \

set_window:     push    bx                      ;Save registers
                push    cx
                push    dx
                mov     al,1
                cmp     ch,dh                   ;Check coordinates
                ja      exit
                cmp     cl,dl
                ja      exit
                mov     upper_left,cx           ;Update coordinates
                mov     lower_right,dx
                mov     ah,18                   ;Read blank attribute
                int     10h
                mov     ax,600h                 ;Blank entire window
                int     10h
                mov     ah,15                   ;Read current page
                int     10h
                mov     dx,cx                   ;Home cursor
                mov     ah,2
                int     10h
                mov     al,0
exit:           pop     dx                      ;Restore registers
                pop     cx
                pop     bx
                iret

comment         \

                Get window coordinates.

                This function call returns the coordinates of the upper left
                and lower right corners of the current display window.

                Entry:  ah = 17 (function code)
                Exit:   cx = upper left corner
                        dx = lower right corner

                All registers preserved except cx and dx.

                \

get_window:     mov     cx,upper_left           ;Read coordinates
                mov     dx,lower_right
                iret

comment         \

                Get blanking attribute.

                This function call returns the attribute of the character
                at the current cursor position, if in alpha mode, or the
                background color (0) if in graphics mode. The resulting
                attribute is meant to be used when scrolling the screen.

                Entry:  ah = 18 (function code)
                Exit:   bh = blanking attribute

                All registers preserved except bx.

                \

get_blanking:   push    ax                      ;Save registers
                mov     ah,15                   ;Read current page
                int     10h
                xor     ah,ah                   ;Background color
                cmp     al,3                    ;Check for alpha modes
                jbe     alpha
                cmp     al,6                    ;Check for graphics
                jbe     graphics
alpha:          mov     ah,8                    ;Read attribute
                int     10h
graphics:       mov     bh,ah                   ;Return attribute
                pop     ax
                iret

comment         \

                Write TTY.

                This function call replaces the old call of the same name.
                It performs the same functions but allows the current window
                to be user defined instead of the whole screen.

                Entry:  ah = 14 (function code)
                        al = character to write
                        bh = page number to write on
                        bl = foreground color (in graphics modes)

                All registers preserved.

                \

write_tty:      cmp     al,7                    ;Let BIOS ring the bell
                je      bios
```

(continued on next page)

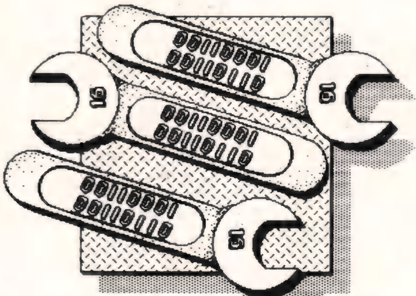
(Listing continued, text begins on page 106.)

```
            push    ax                          ;Save registers
            push    bx
            push    cx
            push    dx
            push    ax                          ;Save character
            mov     ah,3                        ;Read cursor position
            int     10h
            pop     ax                          ;Recover character
;           Check for unprintable control characters

            cmp     al,8                        ;Backspace
            je      bs
            cmp     al,10                       ;Line feed
            je      lf
            cmp     al,13                       ;Carriage return
            je      cr

;           Character is printable

            mov     cx,1                        ;Print character
            mov     ah,10
            int     10h
            inc     dl                          ;Advance cursor
            cmp     dl,right
            jbe     set_cursor

;           Right edge of window exceeded - wrap to next line

            mov     dl,left
lf:         inc     dh
            cmp     dh,lower
            jbe     set_cursor

;           Lower edge of window exceeded - scroll window up

            push    bx                          ;Save page
            mov     ah,18                       ;Read blank attribute
            int     10h
            mov     cx,upper_left               ;Scroll window up
            mov     dx,lower_right
            mov     ax,601h
            int     10h
            pop     bx                          ;Restore page

;           Return cursor to left-hand edge

cr:         mov     dl,left                     ;Carriage return

;           Set cursor to new position

set_cursor: mov     ah,2                        ;Set cursor
            int     10h
            pop     dx                          ;Restore registers
            pop     cx
            pop     bx
            pop     ax
            iret

;           Backspace does not wrap past left edge of window

bs:         dec     dl                          ;Back up
            cmp     dl,left                     ;Past left edge?
            jb      cr                          ;Yes, reset cursor
            jmp     set_cursor                  ;No, leave it alone

interrupt   endp

greeting    db      13,10
            db      218,30 dup (196),191,13,10
            db      179,' The Graphic Software Company ',179,13,10
            db      179,'  BIOS Window Extension V1.1  ',179,13,10
            db      192,30 dup (196),217,13,10,'$'

error_msg   db      13,10,'Invalid window coordinates',13,10,'$'

comment     \

            The install procedure is invoked through the DOS entry point
            when the program is first run. It installs the new interrupt
            and prints a message on the console. When done, it returns to
            DOS and allows the space it occupies itself to be reclaimed.

            The program first tests whether the BIOS extensions are already
            installed. If they are not, this can be detected by the fact
            that a call to an illegal function will return without altering
            any registers.

            \

            assume  ds:code

install     proc    near
            mov     ah,17                       ;Read coordinates
            int     10h
            inc     cl                          ;Alter their value
```

```
            mov     al,cl
            int     10h                             ;Read them again
            cmp     al,cl                           ;Test for difference
            pushf
            jne     installed

;           Install BIOS extensions

            mov     dx,offset greeting              ;Report installation
            mov     ah,9
            int     21h
            mov     ax,3510h                        ;Read old interrupt
            int     21h
            mov     word ptr old_int,bx             ;Save old interrupt
            mov     word ptr old_int+2,es
            mov     dx,offset interrupt             ;Install new interrupt
            mov     ax,2510h
            int     21h

;           BIOS extensions are installed now

installed:  push    cs                              ;Point to command tail
            pop     es
            mov     di,81h
            mov     cx,7fh                          ;Read coordinates
            mov     al,'('
            repne   scasb                           ;Find first pair
            call    num_pair
            push    dx
            repne   scasb                           ;Find second pair
            call    num_pair
            pop     cx
            mov     ah,16                           ;Set coordinates
            int     10h
            or      al,al                           ;Test legality
            jz      legal

;           Window coordinates are illegal

            mov     dx,offset error_msg             ;Print error message
            mov     ah,9
            int     21h

;           Terminate program in appropriate manner

legal:      popf                                    ;Check residency
            jne     resident
            mov     dx,offset greeting              ;Make resident
            int     27h

resident:   int     20h                             ;Already resident

num_pair    proc    near
            push    bx
            call    number                          ;Read first number
            mov     bh,dl
            call    number                          ;Read second number
            mov     dh,bh
            pop     bx
            ret                                     ;Row/Col pair in dx
num_pair    endp

number      proc    near
            push    ax
            push    bx
            mov     al,' '                          ;Skip leading blanks
            repe    scasb
            dec     di
            mov     bl,10                           ;Decimal multiplier
            xor     ax,ax
            xor     dx,dx
digit:      xchg    ax,dx                           ;Multiply by 10
            mul     bl
            add     ax,dx                           ;Add new digit
            xchg    ax,dx
            mov     al,es:[di]                      ;Read next character
            inc     di
            sub     al,'0'                          ;Normalize it
            cmp     al,'9'                          ;Check for digits
            jbe     digit
            pop     bx
            pop     ax
            ret                                     ;Number in dx
number      endp

install     endp

code        ends
            end     DOS_entry                       ;Must be far label
```

End Listing

16-BIT SOFTWARE TOOLBOX



MS DOS Reference Books

Although books on 8086/88 assembly-language programming abound, books on programming in the MS DOS environment have been nearly nonexistent. Several new books on this subject have appeared recently, however, and I would like to mention a few I have looked at:

Simrin, Steven. *MS-DOS Bible*. Indianapolis: Howard W. Sams & Co., 1985. 385 pages with index.

I would characterize this as an introductory systems guide to MS DOS. Exposition of the MS DOS hierarchical file structure, I/O redirection and pipes, installation of a fixed disk, batch files, the MS DOS intrinsic and extrinsic commands, and use of *EDLIN*, *LINK*, and *DEBUG* comprise the majority of the book. Two brief chapters discuss disk formats, directories, the file allocation table, and device drivers; the actual MS DOS function calls are relegated to an appendix. The book contains virtually no programming examples.

Jump, Dennis N. *Programmer's Guide to MS-DOS*. Bowie, MD: Brady Communications Co., 1984. 244 pages with index.

Although this book carries a 1984 copyright date, I did not see it in the bookstores until a few months ago. It is aimed at beginning assembly-language programmers and discusses the various MS DOS function calls by category (traditional character I/O calls, traditional file management calls, extended file management calls, and so on). There is minimal coverage of MS DOS loading, structure, or disk control areas. The explanations of memory management, the MS DOS *EXEC* call, and installable device drivers are reasonable though incomplete. The last 30 or so pages of the book are devoted to IBM PC-specific topics, mainly the ROM BIOS driver calls. Programming examples are chiefly in the form of assembly-language functions to be called from Pascal, except for an example character device driver. This book, together with the *MS-DOS Bible* by Simrin would be a good starting point for beginning MS DOS assembly-language programmers as the two books have little overlap in content.

by Ray Duncan

King, Richard A. *The MS-DOS Handbook*. Berkeley, CA: Sybex Computer Books, 1985. 319 pages with index.

This book is written for beginning assembly-language programmers; it covers most of the same material as the previous two books combined except that disk structures and control areas are discussed in a rather general way and device drivers are barely mentioned. It contains few programming examples. My biggest complaint about this book is that material applicable to generic MS DOS systems and material pertinent only to IBM PCs is jumbled together with very little distinction.

Norton, Peter. *The Peter Norton Programmer's Guide to the IBM PC*. Bellevue, WA: Microsoft Press, 1985. 426 pages with index.

As you would expect, this book is heavily slanted toward the IBM PC family with extensive discussion of the IBM video adapter, keyboard, sound generation, and ROM BIOS. It may best be viewed as a well-organized, highly readable replacement for both the *IBM PC DOS Technical Reference* and the *IBM PC Hardware Reference* manuals. It's directed at programmers of intermediate experience.

Information on the generic MS DOS services occupies less than a quarter of Norton's book but is still relatively thorough; additional quick-reference tables are well laid out. The book has clear explanations of the MS DOS disk structure and control areas, file control blocks, and program segment prefixes. The section on the *EXEC* function is inadequate to make it work (unless the reader already knows how) because the special use of the modify memory block function *4ah* with register *ES* pointing to the calling program's PSP is not detailed; also, the author's comment on registers destroyed by *EXEC* and the method of saving and restoring the *SS* and *SP* registers are not correct. Coverage of device drivers is limited to some discussion and (deserved) criticism of ANSI.SYS; there is no material on the structure or programming of device drivers. Assembly-language examples are sparse and simplistic and are nearly all from the point of view of Pascal or C subroutines.

Lafore, Robert. *Assembly Language Primer for the IBM PC and XT*. New York: New American Library, 1984. 501 pages with index.

Like the Norton book, this book is strongly biased toward the IBM PC hardware and ROM BIOS software, but about a quarter of it is devoted to discussion of general MS DOS topics. The layout of DOS and the structure of executable program files are discussed briefly. File I/O is well covered, but advanced subjects such as device drivers, memory allocation, and the *EXEC* call are completely absent. This book makes extensive use of hex memory dumps and assembly-language examples, including complete working programs—a style that should be commended.

MS-DOS Programmer's Reference.

Bellevue, WA: Microsoft Press, 1984.

This book is rarely seen in its original Microsoft form. It is, however, commonly available in various MS DOS OEM editions, the most prevalent (of course) being the *IBM PC DOS Technical Reference*. Versions are also extant from Hewlett-Packard, Zenith, and Intel, among others. The Intel book is the best of the bunch; it has been extended with assembly-language examples for each MS DOS system function, source code for simple block and character device drivers, and a lengthy appendix describing the Intel relocatable object record format. It can be ordered from the Literature Department, Intel Corp., 3065 Bowers Ave., Santa Clara, CA 95051 (part number 135555-001).

DDJ readers are invited to send further comments on the above books to this column and to provide information on other books they have found useful. Two additional books on MS DOS programming, aimed at intermediate to advanced assembly-language programmers, are due from Microsoft Press about the time this column will appear in print. The first is my own book *Advanced MS-DOS*. The other is the *MS-DOS Technical Reference Encyclopedia*, Volume I (collaboration of several authors, preface by Bill Gates, about 1200 pages, $134.95, June publication date expected).

The EXEC Function and FORTRAN

Chris Dunford, author of the popular CED command line editor, sent a comment on the interface between FORTRAN and the *EXEC* call that was printed in the January 1986 16-Bit Toolbox: "I wonder if you're aware that there is an easier way to find the PSP of a program: DOS function *62h* returns (in register *bx*) the PSP address of the currently executing process. Function *62h* is not available under DOS 2.x, but undocumented function *51h* performs the same service. You want to avoid using undocumented features, of course, but I don't foresee new releases of DOS 2.X in the near future!

"It seems to me quite safe to check the DOS version under which your program is running, then use function *51h* or *62h* as appropriate. Of course, another advantage of this method over Sybek's is that it's compiler independent."

The 20 Files Problem

Dan Daetwyler's description of his problem with DOS' limit of 20 open files per process, which appeared in the December 1985 16-Bit Toolbox column, provoked several helpful responses from *DDJ* readers.

Sean McDowell of Marietta, Georgia, said: "I am writing in response to Mr. Daetwyler's letter about DOS 3.0 and the 20 file handle limit. We have had this problem since handles were first introduced. I am very curious about why Mr. Daetwyler didn't run into this problem sooner. The problem seems to stem from the amount of room available in the program segment prefix (PSP). If you look at the memory map of the PSP in the *IBM Technical Reference*, there is an area from offset *18h* to *2bh* marked as reserved. This appears to be where DOS stores the allocated file handle numbers. Because there are 20 bytes, there are 20 file handles for the cur-

(continued from page 107)

rent process. DOS seems to use the segment address of the current PSP (see DOS 3.0 function *62h*) as the process ID. Because an interrupt to a resident database-handling module would not change this value, that would explain why splitting the database and allowing the resident task to own part would not correct the problem. . . . Currently we are opening and closing handles with a least-recently-used algorithm within our file access routines, even though that can be painfully slow at times."

Inspecting the PSPs of some of my own programs, it does seem evident that the PSP handle table positions correspond to handle numbers. The byte at a given table position contains 0FFH if the handle is not in use for the process. If the handle is opened to a file or device, the byte contains a small integer number that is probably an index to DOS' internal table of file control blocks for the "extended" file functions.

Ross Nelson of San Jose, California, took this a little further: "The DOS equivalent of the process ID is the PSP paragraph number. This is how DOS keeps track of who owns memory blocks, and I suspect it manages open files in this way as well. It keeps an internal variable to track the 'currently active PSP,' and there are two (undocumented naturally) calls to get/set this variable.

"Executing an *interrupt 21h* with *ah=51h* will return the currently active PSP in the *bx* register. Loading *bx* with a PSP and executing an *interrupt 21h* with *ah=50h* will set a new active PSP. The pseudotasking solution to the file problem is then handled in this way:

Task A: open initial files, do processing, *EXEC* Task B
Task B: open new files, call (via software interrupt?) to Task A
Task A: save active PSP, set active to Task A's PSP, do processing, restore previous PSP, return
Task B: continue processing. . ."

An unsigned note from another reader, along with a labeled hexadecimal dump of a program segment prefix, also took Mr. McDowell's observation on the PSP a little further. The anonymous reader pointed out that the words at *PSP+0032h*, *0034h*, and *0036h* contain the maximum number of handles allowed for the process, the offset of the handle table, and the segment of the handle table, respectively. The note said: "To open more than 20 handles, provide an area in your program consisting of one byte per handle. Move the first five handles from the table at *PSP+0018h* to the new handle table. Then, change the handle table pointers at *PSP+0034h* to point to the new table, and change the value at *PSP+0032h* to the number of entries in the new table."

PC/AT Interrupts

Brett Salter of Data Base Decisions wrote: "In regard to the AT problem you mentioned in your May 1985 column, I'm afraid Intel's nasty habit of referring to interrupt numbers as decimal instead of hex has caused some confusion. *Interrupt 13* decimal (not hex) is the segment overrun exception interrupt. Because this interrupt (*0dh*) was not previously used, it won't cause anything as disastrous as zapping a hard disk.

"IBM has most of the new exception interrupts vectored to a routine that sets a flag and then *IRET*s back to the user's program. On return, the instruction that caused the exception is reexecuted, putting the system into an enabled loop. The only way out on a normal PC/AT is to use Ctrl-Alt-Del to reboot.

"If you have my Periscope debugger installed on an AT, interrupts *0dh* (segment overrun) and *06h* (invalid opcode) point to Periscope. This means that the instruction sequence shown in your column will put the user into Periscope, where you can modify the registers as needed to recover the system.

"Another new twist on the PC/AT—if your program uses the array *BOUND* instruction, an exception causes *INT 5* to be performed, printing the screen to a parallel printer. Then control returns to the *BOUND* instruction that caused the interrupt, which prints the screen over and over until you reboot the system. Once again, Periscope can be set to intercept this interrupt, letting you regain control of the system."

Macro Assembler, Version 4.0

The new speedy Version 4.0 of the Microsoft Macro Assembler also has a completely new bug! It seems that the *include* function has been changed in such a way that it ignores end-of-file marks (*1ah*) and uses the size of the file as recorded in the disk directory instead. If you use an editor (such as WordStar in nondocument mode), which rounds the size of the file up to the next block boundary and pads the last block out with EOF marks, you will get an "extra characters on line" error for the *include* statement. The listing file cannot be typed or printed past the point of the error because the macro assembler copies all of the extra EOF marks into the listing file! A work-around suggested by Microsoft is to invoke *EDLIN* with the name of the source file, then enter the E command to immediately exit from the line editor. *EDLIN* will scan for the first EOF mark and create a new file with the correct size, renaming the original file with a BAK extension.

IBM PC Window Control

John J. Seal, of the Graphics Software Co., contributed a resident window manager for the IBM PC to this month's 16-Bit column (see Listing One, page 102). The program is invoked with a command in the form:

```
C>WINDOW (R,C) TO (R,C)
```

which specifies a window on the screen in which all subsequent activity will take place. The window coordinates are given as the upper-left and lower-right row and column. Coordinate (0,0) is the upper-left corner of the screen.

WINDOW makes itself resident and captures the IBM BIOS video driver interrupt, filtering out some calls and passing the others on to the ROM driver. The areas outside the current window may be written to directly by any of the usual commands (in your favorite language) that allow direct coordinate specification; the WINDOW program affects only characters that are displayed with the "TTY Output" ROM BIOS call (function *0eh*). The WINDOW program illustrates several useful PC DOS programming techniques:

- installation of a new resident process with the *terminate and stay resident* function
- chaining on to an existing interrupt handler
- using the MS DOS *get interrupt* and *set interrupt* functions to inspect and modify the hardware interrupt table

Although I have not changed Mr. Seal's code, it should be noted that the preferred method to terminate and stay resident for DOS, Version 2 and later, is use of *interrupt 21h* function *31h*, rather than *interrupt 27h*. Similarly, the preferred method of final exit is now *interrupt 21h* function *4ch*, rather than *interrupt 20h*.

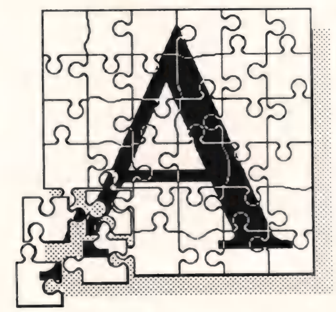
DDJ

(Listing begins on page 102.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8**.

THE RIGHT TO ASSEMBLE

Code Compression with Mini-Interpreters



Code crunching is an arcane art, something like squeezing elephants into phone booths. When I was a game designer at Atari, I constantly had to squeeze my 6502 code down further and further to fit into a pitifully small 4K ROM space. Even after Atari invented a bank-switching ROM cartridge, we still came up against the memory limit very quickly. One of our sayings was that it's always possible to squeeze another byte out of an assembly-language program. Some of the feats we accomplished were truly remarkable—the world's smallest chess program (as far as I know) runs on the 2600 VCS game machine with only 4K of ROM and 128 bytes of RAM! When I left Atari, I thought I would never have to worry about crunching my code again. I was entering the world of "big" machines, with huge 64K RAM spaces! That was at best a very temporary release, though. Soon I was dealing with truly "enormous" programs (operating systems), and once again I had to start crunching my code.

Some of the techniques for effective code crunching are actually quite simple—for example, variables can frequently be moved into low memory, where most processors can access them with fewer bytes of address information. On the 6502 this is a particularly large saving—an instruction that loads a byte from memory location $1000 requires 3 bytes, but if you load the information from $10 you use up only 2. Another

by Nick Turner

trick is to use register variables wherever possible because register instructions are usually shorter. On the first crunching pass, though, all these simple tricks are usually used up. At Atari, this first pass usually reduced the size of the game program by no more than 15 percent. Because the "raw" programs were about twice as large as the space into which they had to be shoehorned, there was still a lot of crunching to be done.

Using a Mini-Interpreter

The technique I'll talk about here was used with some success in many of my operating system designs. It's best applied when you have a lot of code that contains repetitive calls to several different subroutines. In one particular case, I was writing a system that would be accessed over the phone by users with serial terminals. It was important to be able to transmit various sequences of control characters and also to be able to receive and interpret similar sequences, setting and clearing various flags and changing the modes of input and output. To do this with traditional code would have worked, but I managed to chop the size of the required code by more than half using what I call a mini-interpreter.

A mini-interpreter is a subroutine that pops the stack to find out where it was called from, then reads the bytes immediately following the calling instruction and performs various tasks based on what is sees there. At some point it will see a *return* token and will perform a jump to the next valid instruction following. In the case of a machine in which the instructions must be word-aligned, such as the 68000, it would return to the next even-addressed byte.

The mini-interpreter I wrote for my dial-up system was originally created as a print routine that would read an ASCII string immediately after the invoked instruction, then print it out. A null ($00) was used to terminate the string. (A 6502 routine by Chris Espinoza that does just this was published in the September 1976 issue of *DDJ*.) Here's a typical calling sequence for the routine as described so far:

```
...                     ;code preceding the call
JSR     PRINT           ;call the print routine
ASC     "This text would be printed out."
HEX     00
...                     ;code following the call
```

Note that the calling sequence requires 4 bytes; 3 for the *JSR* instruction and 1 for the null-token terminator. I felt that this 4-byte overhead was excessive. How could I shave it down?

The answer occurred to me one day while I was setting a breakpoint to debug a program: The *BRK* instruction, which is used by debuggers to wrest control from the executing program when it reaches a $00 in the program, occupies only 1 byte and can cause execution to vector to any point in memory. The only problem was that the *BRK* instruction was used extensively during debugging. So, I created a conditional macro that would assemble the *PRINT* call as a *JSR* or a *BRK*, depending on whether I was debugging or not. Then I added a small routine in the initialization that placed the address of my *PRINT* routine into the *BRK* vector. Of course, the *PRINT* routine itself also had to have some conditionally assembled code because the *BRK* instruction puts some status information on the stack that I needed to pop and ignore. Only a few extra instructions were needed, however. The new method worked beautifully. Now I could

print any text I wanted with only a 2-byte overhead. Of course, the routine that handled the *PRINT* calls took up some space of its own, but with dozens of calls in my code, the extra overhead was more than compensated for by the code savings in my *PRINT* calls.

Further Refinements

I was quite pleased with the results, but I still needed to shave the program down further. I discovered that there were about a dozen macro expansions that frequently occurred adjacent to the *PRINT* calls. Many of them were used to set, test, or clear various print format flags—for example, there was a protocol with which the user could send a Control-C to inhibit furthur output until the beginning of the next logical block of text. To implement this protocol, I created an output-inhibit flag and tested the status of the flag before sending each character. The flag was reset at the beginning of each block of text and set whenever a Control-C was detected from the user. The problem was that the flag reset often occurred in the middle of a stream of text, requiring a code sequence something like this:

```
...                      ;text being output
ASC     "The last text in the first
                               block."
HEX     00               ;marks end of text
LDA     TXTFLAG          ;clear output sup-
        TXTFLAG               press flag
AND     #255-OUTSUP
STA     TXTFLAG
PRINT                    ;macro that in
                         vokes print inter-
                         preter (JSR or BRK)
BRK
ASC     "First text in the next block."
...                      ;more text
```

Look at all those bytes! Including the end token of the previous block, the code to clear the flag, and the *BRK* to call the interpreter again, I used up 8 bytes! How could I cut that down?

The answer became obvious as soon as I realized that the text-output interpreter could easily be increased in size without adding very much to the overall size of the program. Because there were quite a few 8-bit values that could never be encountered in the middle of a text stream, I simply appropriated one, equated it to *CLCTLC* (CLear ConTroL C), and stuck it into the text stream. After adding the appropriate code to the interpreter, I reduced the above monstrosity to:

```
...                      ;text
ASC     "The last text in the first
                               block."
BYTE    CLCTLC           ;clear output
                             suppression
ASC     "First text in the next block."
...                      ;more text
```

This was the start of something big (or, rather, something small!). There were several other flags whose entire manipulation could be compressed in the same way. Eventually, I added a lot more to the routine. I added escape codes that would switch to a different interpreter altogether. I added a set of conditional test codes that allowed me to output one of two strings depending on the state of various flags. I even added a code that allowed me to call any machine-language subroutine anywhere and then return to interpreted execution upon encountering the *RTS* from the called routine (an interpreted *JSR* instruction). This one was particularly useful because it allowed me to perform special-purpose routines without the code overhead of leaving the text output interpreter.

I never ran into any difficulties with this heavily interpreted approach, and I was able to reduce the code overhead vastly for "stupid little stuff" that I had to do repeatedly. If your application is time critical, you might not want to use a mini-interpreter. But because code size was far more important to me than the time required to execute the instructions, this approach was perfect for my on-line system, which you can dial up today at (408) 338-9511.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9**.

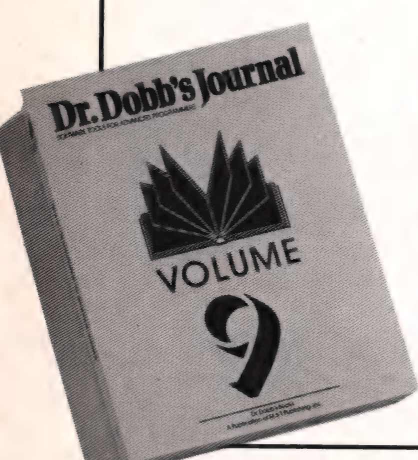
Dr. Dobb's Catalog

Continuing the tradition of providing its readers with useful, powerful software tools, Dr. Dobb's Journal of Software Tools presents this collection of programming tools in the form of books and software on disk.

Announcing for May...

Dr. Dobb's Bound Volume 9

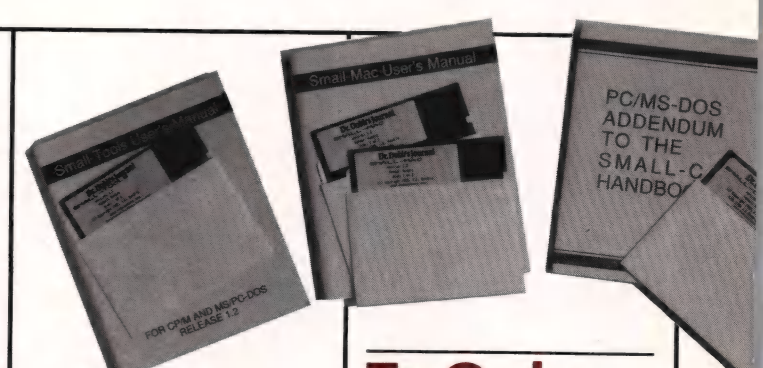
This ninth volume in the ***Dr. Dobb's Bound Volume*** series contains the complete editorial contents of *Dr. Dobb's Journal of Software Tools* for 1984: over 1000 pages of text and code. To commemorate the release of Volume 9, we are offering a discount on the full set of Bound Volumes. This set includes every issue of *DDJ* from 1976 through 1984. If you order the set now, you will receive a $40 discount.



DDJ Listings on Disk

You've demanded it, and we're delivering.

Dr. Dobb's Journal of Software Tools now offers a listings service. The first disk has just been released, containing most of the program listings that appeared in the magazine from January through April 1986. This accompaniment to the magazine is available in various formats.



Also Inside

- **Dr. Dobb's Complete Toolbox of C**
- **A Unix-like Shell for MS DOS** and
- **A Unix-like Utility Package for MS DOS** by Allen Holub
- **Dr. Dobb's Z80 Toolbook** by David Cortesi

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or

CALL TOLL-FREE
1-800-528-6050 EXT. 4001

and refer to product item number, title, and disk format.

For customer service questions,
CALL M&T PUBLISHING, INC.
415-366-3600 EXT. 209

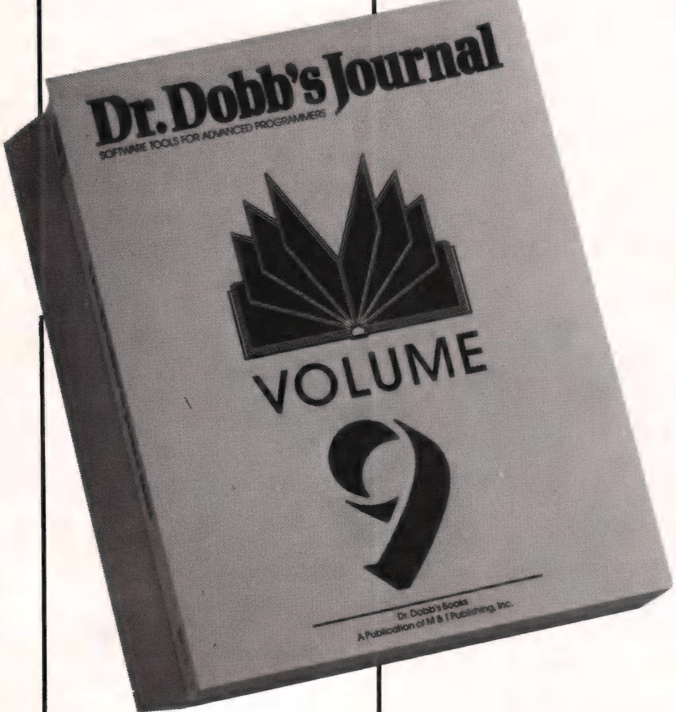
M&T BOOKS

Dr. Dobb's Bound Volumes

Programs by the Pound

Announcing: Dr. Dobb's Bound Volume 9

Over 1000 pages of listings and text. The entire 1984 editorial contents of *Dr. Dobb's Journal of Software Tools.*



Bound Volume 9: 1984

Item #020B

Shaping things to come. In 1984 new Editor-in-chief Mike Swaine brought his interests in advanced technology to *Dr. Dobb's Journal.* We presented the concepts behind Prolog and published an expert system for weather prediction. We learned Modula-2, and taught Forth to talk to a 68000, to MS DOS, and to the people of China. We examined a new language implementation called Turbo Pascal and extended implementations of C with a preprocessor, a library, Tony Skjellum's tricks, and Allen Holub's Grep. We published two powerful encryption systems, telecommunications protocols, floating-point benchmark results, and an issue devoted to the internals of Unix. And Ray Duncan, Bob Blum, and Dave Cortesi were on hand with their fascinating columns.

Bound Volume 1: 1976

Item #013

The working notes of a technological revolution. Programmers from Defense laboratory systems analysts to kitchen-table entrepreneurs worked for the intrinsic rewards to put development software on the brand-new invention, the microcomputer. Before there was an Apple, *Dr. Dobb's Journal of Tiny Basic Calisthenics and Orthodontia* (subtitle: Running Light without Overbyte) was founded to put a programming language on the machines, and became both chronicler and instrument of the revolution. In this first-year volume: Tiny Basic, the first word on CP/M, notes on building an IMSAI, floating-point and timer routines.

Bound Volume 2: 1977

Item #014

Running light without overbyte. By year two, *Dr. Dobb's* formula was concocted: tough questions and serious technical issues handled with enthusiasm, wit, and scant reverence for the accepted answers. Source code. Tools for programmers. Respect for tight programming. *Dr. Dobb's Journal* readers shared insights on warping the Intel 8080 into a computer CPU, and *Dr. Dobb's* published a complete operating system for the chip. A motley crop of computers and software products were popping up, and *Dr. Dobb's* investigated: the Heath H-8, the KIM-1, the Alpha Micro, MITS Basic, Poly Basic, and Lawrence Livermore Labs Basic. *Dr. Dobb's* introduced Pilot for microcomputers and published tips on doing string handling, high-speed I/O, and turtle graphics in limited memory.

Bound Volume 3: 1978

Item #015

The roots of Silicon Valley growth. In 1978 Steve Wozniak and other programmers were publishing in *Dr. Dobb's Journal* code that would help them grow multi-million-dollar computer companies. The proposed S-100 bus standard was hashed out in *Dr. Dobb's* pages. *Dr. Dobb's* contributors began to speak more in terms of technique than of specific implementations as the industry began to diversify. Languages covered in depth included SAM76, Pilot, Pascal, and Lisp.

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or

CALL TOLL-FREE
1-800-528-6050 EXT. 4001
and refer to product item number, title, and disk format.

For customer service questions,
CALL M&T PUBLISHING, INC.
415-366-3600 EXT. 209

M&T BOOKS

Dr. Dobb's Bound Volumes

Programs by the Pound

Bound Volume 4: 1979

Item #016

In the midst of the Gold Rush. Three years before IBM would release its PC, a thriving, rough-and-tumble personal computer industry existed. Fortunes had been made and lost, the effective power of the machine multiplied a hundredfold. By 1979 some stability had even emerged; one could speak of the processors that had proven longevity as microcomputer CPUs: the 8080, the Z80, the 6800, and the 6502. *Dr. Dobb's Journal* focused on the best ways to use these processors, with algorithms, tips, and code for 8- to 16-bit conversion, pseudo-random number generation, micro-to-mainframe connections, telecommunications, and networking. And lots of useful code.

Bound Volume 5: 1980

Item #017

The preeminence of CP/M and the rise of C. More than any other magazine, *Dr. Dobb's Journal* was responsible for the spread of CP/M and C on microcomputers. Both of those movements began in 1980. *Dr. Dobb's* all-CP/M issue, including Gary Kildall's history of CP/M, sold out within weeks of publication. This was the year of Ron Cain's original Small C compiler, of a CP/M-oriented C interpreter, CP/M-to-UCSD Pascal file conversion techniques, and of a greater concern in *Dr. Dobb's* with software portability.

Bound Volume 6: 1981

Item #018

The first of Forth. 1981 saw *Dr. Dobb's* first all-Forth issue (now sold out), along with an emphasis on CP/M, C, telecommunications, and new languages. David Cortesi began "Dr. Dobb's Clinic," one of the magazine's most popular features. Highlights included information on PCNET, the Conference Tree, the Electronic Phone Book, Tiny Basic for the 6809, writing your own compiler, and a systems programming language.

Bound Volume 7: 1982

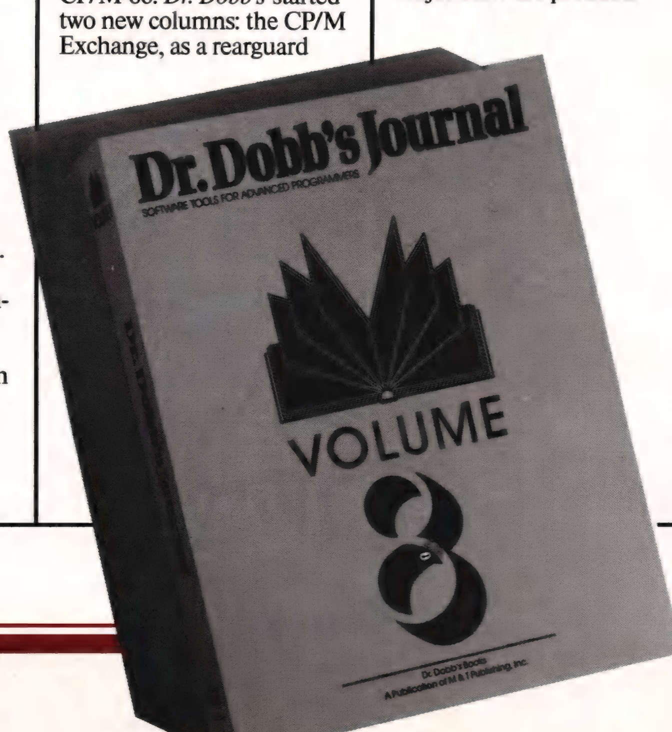
Item #019

Legitimacy. By 1982 IBM had become a player in the personal computer game and was changing the rules. New microprocessors arrived, the first designed specifically to serve as personal computer CPUs. In *Dr. Dobb's Journal* Dave Cortesi published the first serious comparison of MS DOS and CP/M-86. *Dr. Dobb's* started two new columns: the CP/M Exchange, as a rearguard maneuver to ensure that good tools for CP/M programmers would continue to be developed and circulated, and the 16-Bit Software Toolbox to investigate the 8088/86 and other new microprocessors. We published code for the 68000 and Z8000 processors, and looked ahead, in a provocative essay, to fifth-generation computers.

Bound Volume 8: 1983

Item #020

Power tools. Personal computers were proving themselves to be true professional software development tools by 1983, the year in which Jim Hendrix completed his "canonical" version of Small C in *Dr. Dobb's Journal*. *Dr. Dobb's* published more 68000 and 8088 code, and as the memory limitations relaxed, the magazine's commitment to tight code let it shoehorn impossibly large systems into memory. Small C was just one of the major software products published in their entirety in *Dr. Dobb's* pages that year; there were Ed Ream's RED screen editor and a version of the Ada language called Augusta.



Buy the complete set and save 15%

If you buy all nine volumes, covering the entire editorial content of ***Dr. Dobb's Journal of Software Tools*** from the first issue in 1976 through 1984, you pay just $225. That's a 15% discount and over $40 off the combined individual prices. To order the complete set of Bound Volumes 1 through 9, ask for item #020C.

Item #013	$27.75	Vol. 1
Item #014	$27.75	Vol. 2
Item #015	$27.75	Vol. 3
Item #016	$27.75	Vol. 4
Item #017	$27.75	Vol. 5
Item #018	$27.75	Vol. 6
Item #019	$30.75	Vol. 7
Item #020	$31.75	Vol. 8
Item #020B	$35.75	Vol. 9

All 9 volumes
Item #020C $225.00

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or

CALL TOLL-FREE
1-800-528-6050 EXT. 4001
and refer to product item number, title, and disk format.

For customer service questions,
CALL M&T PUBLISHING, INC.
415-366-3600 EXT. 209

M&T BOOKS

A collection of powerful tools for C software developers, including books, software on disk, and reference materials from the publisher of Dr. Dobb's Journal of Software Tools

Dr. Dobb's Complete C Toolbox

From M&T Publishing and Brady Communications...

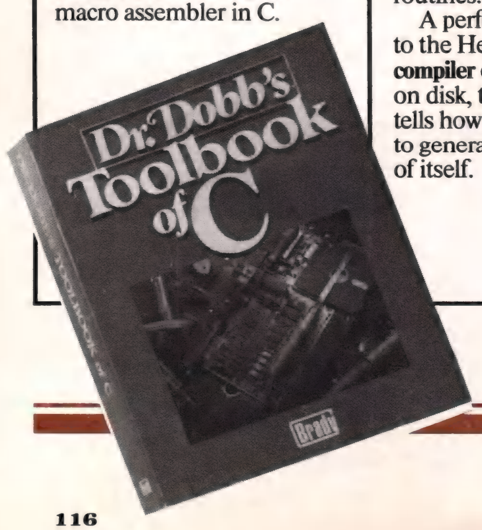
Dr. Dobb's Toolbook of C

Item #005

The **Toolbook** contains over 700 pages of C material, including articles by such C experts as Kernighan and Ritchie, Cain and Hendrix, Skjellum and Holub. The level is sophisticated and pragmatic, appropriate for professional C programmers.

The most valuable part of the **Toolbook** to many will be the hundreds of pages of useful C source code, including a complete compiler, an assembler, and text-processing utilities. The accompanying text explains, in the programmers' own words, why they did what they did.

Dr. Dobb's Journal of Software Tools introduced a generation of personal computer programmers to the C programming language, and all the best C articles and code published in *Dr. Dobb's* over the years is included and updated in the **Toolbook**, including Ron Cain's original Small C article and articles from sold-out issues. But the **Toolbook** also includes material never before published, including Jim Hendrix's complete macro assembler in C.



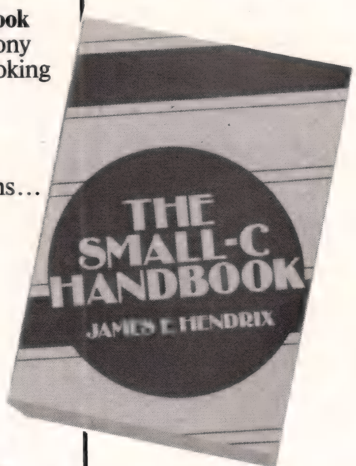
Dr. Dobb's offers the **Toolbook** in a **special hard-bound edition** for just $29.95. You'll find:

Jim Hendrix's famous **Small C Compiler** and **New Library for Small C** (both also available on disk),

NEW: Hendrix's **Small Mac: An Assembler for Small C** and **Small Tools: Programs for Text Processing** (both also available on disk),

All of Tony Skjellum's **C Programmer's Notebook** columns distilled by Tony into one thought-provoking chapter.

Also from M&T Publishing and Brady Communications...



The Small C Handbook

Item #006 or #006A

Jim Hendrix's **Small-C Handbook** is the reference book on his Small-C compiler. In addition to describing the operation of the compiler, the book contains complete source listings to the compiler and its library of arithmetic and logical routines.

A perfect companion to the Hendrix **Small-C compiler** offered by *Dr. Dobb's* on disk, the **Handbook** even tells how to use the compiler to generate a new version of itself.

While both the **Handbook** and the **Toolbook** provide documentation for the Small-C compiler, the **Handbook** contains a more detailed discussion and is available with an addendum for the MS/PC DOS version.

The **Handbook,** Item #006, is just $17.95. Jim Hendrix has ported the compiler to the MS/PC DOS environment since the **Handbook** was printed, and the **Handbook** plus his **MS/PC DOS Handbook Addendum,** Item #006A, is $22.95.

Dr. Dobb's C Software Tools on Disk

To complement the **Toolbook,** *Dr. Dobb's* also offers the following programs on disk. Full C source code and documentation is included. Except where indicated, both CP/M and MS/PC DOS versions are available.

While both the **Handbook** and the **Toolbook** provide documentation for the Small-C compiler, the **Handbook** contains a more detailed discussion and is available with an addendum for the MS/PC DOS version. The **MS/PC DOS Small-C Handbook Addendum** is recommended in addition to the **Handbook** for MS DOS or PC DOS users.

The **Small-C Compiler** is available for $19.95 in either the CP/M or the MS/PC DOS version.

Small-C Compiler

Item #007

Jim Hendrix's Small-C Compiler is the most popular piece of software ever published in *Dr. Dobb's* 10-year history. Like a home-study course in compiler design, the **Small-C Compiler** and **Small-C Handbook** provide everything you need but the computer for learning how compilers are constructed, and for learning C at its most fundamental level.

To Order:

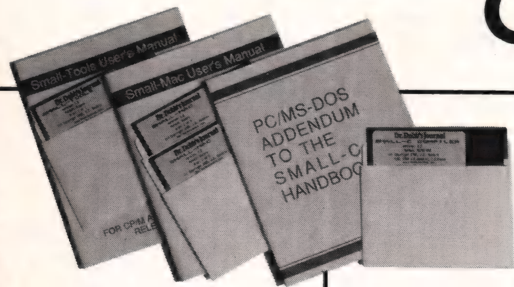
To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or

CALL TOLL-FREE
1-800-528-6050 EXT. 4001
and refer to product item number, title, and disk format.

For customer service questions,
CALL M&T
PUBLISHING, INC.
415-366-3600 EXT. 209

M&T BOOKS

Dr. Dobb's Complete C Toolbox



Small-Mac: An Assembler for Small-C

Item #012A

Small-Mac is an assembler designed to stress simplicity, portability, adaptability, and educational value. The package features a simplified macro facility, C language expression operators, object file visibility, descriptive error messages, and an externally-defined machine instruction table. You get the macro assembler, linkage editor, load-and-go loader, library manager, CPU configuration utility, and a utility to dump relocatable files.

Small-Mac is available with documentation for $29.95. For CP/M systems only.

Small-Tools: Programs for Text Processing

Item #010A

A package of programs performing specific, modular operations on text files, including: editing; formatting; sorting; merging; listing; printing; searching; changing; transliterating; copying; concatenating; encrypting and decrypting; replacing spaces with tabs and tabs with spaces; counting characters, words, or lines; and selecting printer fonts. Supplied in source code form so you can select and adapt these tools to your own purposes.

Small-Tools is available with documentation for $29.95. For CP/M or MS/PC DOS systems.

Special Packages — 20% Off

Now for almost 20% off the combined individual product prices, you can order a complete set of *Dr. Dobb's* C programming tools for your CP/M or MS/PC DOS system.

C Package for CP/M

Item #005A

Ordered individually, these items would cost about $120. If you order the CP/M C package, you'll get ***Dr. Dobb's Toolbook***, the **Small-C Handbook**, the **Small-C Compiler** on disk, the **Small-Mac** assembler on disk with documentation in the **Small-Mac Manual**, the **Small-Tools** text-processing programs on disk with documentation in the **Small-Tools Manual**, all for just $99.95.

C Package for MS/PC DOS

Item #005B

These items would cost over $100 if purchased individually. If you order the MS/PC DOS C package, you'll get ***Dr. Dobb's Toolbook***, the **Small-C Handbook** with the **MS/PC DOS Addendum**, the **Small-C Compiler** on disk, the **Small-Tools** text-processing programs on disk with documentation in the **Small-Tools Manual**, all for just $82.95.

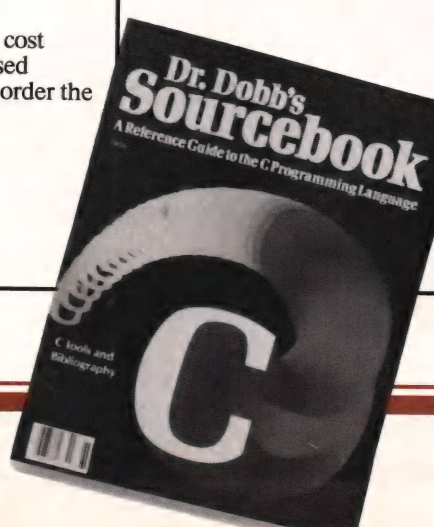
Dr. Dobb's Sourcebook: A Reference Guide to the C Programming Language

Item #004

Products and services for C programmers are appearing at so rapid a rate that it's all but impossible to keep up with them. *Dr. Dobb's* presents this handy guide to the who, what, when, where, and why of C. A comprehensive reference manual to new information, products, and services, the **Sourcebook** contains:

- A bibliography of over 300 articles and books on the C language;
- A descriptive list of products for C programmers: compilers, editors, interpreters, and utilities;
- A list of C-related services: classes, seminars, and on-line services.

The **Sourcebook** is just $7.95.



Dr. Dobb's Sourcebook: A Reference Guide to the C Programming Language Item #004	$ 7.95
Dr. Dobb's Toolbook for C Item #005	$29.95
The Small-C Handbook Item #006	$17.95
The Small-C Handbook and MS/PC-DOS Addendum Item #006A	$22.95
Small-C Compiler disk Item #007	$19.95
Small Tools: Programs for Text Processing disk Item #010A	$29.95
Small Mac: An Assembler for Small-C disk (For CP/M only.) Item #012A	$29.95
CP/M C Package Item #005A	$99.95
MS/PC DOS C Package Item #005B	$82.95

For CP/M disks, please specify one of the following formats: Apple, Osborne, Kaypro, Zenith Z-100 DS/DD, 8″ SS/SD.

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or

CALL TOLL-FREE
1-800-528-6050 EXT. 4001

and refer to product item number, title, and disk format.

For customer service questions,
CALL M&T PUBLISHING, INC.
415-366-3600 EXT. 209

M&T BOOKS

Dr. Dobb's Listings On Disk

**Finally!
You've Asked For It For
Years and Here It Is!**

Dr. Dobb's Listings On Disk

Dr. Dobb's Journal of Software Tools has always provided its readers with valuable code. Now, as a useful adjunct to the magazine, DDJ offers the additional value and convenience of selected listings on disk! ***Dr. Dobb's Listings #186*** is a collection of listings from Dr. Dobb's January 1986 issue, through the April 1986 issue. In this first of three ***Dr. Dobb's Listings*** disks for 1986 you'll find listings from the following DDJ articles, among others.

From Issue #111 January 1986

***A Simple OS for Real-time Applications; 68000 assembly language techniques for an operating system kernel* by DDJ editor Nick Turner

***Exec calls and Fortran; a technique allowing execution of a user or system task from a user program* from DDJ's 16-Bit Software Toolbox, by Robert Sypek

***32-bit Square Roots; An 8086 assembly-language routine for 32-bit square roots* by Michael Barr

From Issue #112 February 1986

***Fast Integer Powers for Pascal; An implementation of the fastest-known algorithm for the computation of integer powers* by Dennis E. Hamilton

***Data Abstraction with Modula-2; Construction of a priority queue in Modula-2* by Bill Walker and Stephen Alexander

***Learning Ada on a Micro; A draw poker program in Ada* by Do-While Jones

***Fast IBM PC graphics routines* from DDJ's 16-Bit Software Toolbox, by Dan Rollins

From Issue #113 March 1986

***Recursive Bose-Nelson Sort; An alternative to Joe Celko's September 1985 sort routine* by R. J. Wissbaum

***A Variable-Metric Minimizer; A C program for minimizing arbitrary functions* by Joe Marasco

***Concurrency and Turbo Pascal; An approach to implementing coroutines in Pascal* by Ernest Bergmann

***Speeding MS DOS Disk Access; Programs to test disk-access speed* by Greg Weissman

***Square Roots on the NS32000; Comparable square root routines in C and assembly language for National Semiconductor's 3200 family* by Richard Campbell

From Issue #114 April 1986

***Boca Raton Inference Engine; Lisp, Prolog, and Expert-2 techniques and code* by Robert Brown

Item #170 $14.95
Dr. Dobb's Listings #186
Please specify MS/DOS, Macintosh, or CP/M. For CP/M disks, please specify one of the following formats: Apple, Osborne, Kaypro, Zenith Z-100 DS/DD, 8" SS/SD.

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or
CALL TOLL-FREE
1-800-528-6050 EXT. 4001
and refer to product item number, title, and disk format.
For customer service questions,
CALL M&T PUBLISHING, INC.
415-366-3600 EXT. 209

M&T BOOKS

[redacted]-like Shell and Utility Package for MS-DOS

Only $29.95 each! Both for only $50 — save over 15%!

Includes complete source code and documentation.

A UNIX-like Shell for MS-DOS and A UNIX-like Utility Package for MS-DOS

by *Dr. Dobb's* C-Chest columnist, ALLEN HOLUB

The Shell

A MS-DOS implementation of the most often used parts of the UNIX C shell. This package includes an executable version of the shell, along **with complete C source code** and full documentation.

Supported features are:

Editing Command line editing with the cursors is supported. The line is visible as you edit it.

Aliases Can be used to change the names of commands or as very fast memory resident batch files.

History The ability to execute a previous command again. The command can be edited before being executed.

Shell Variables Macros that can be used on the command line.

Nested batch files A batch file can call another batch file like a subroutine. Control is passed to the second file and then back to the first one when the second file is finished. DOS doesn't have this capability.

UNIX-like syntax Slash (/) used as a directory separator, minus (−) as a switch designator. A 2048 byte command line is supported. Command line wild card expansion. Multiple commands on a line.

The shell also supports redirection of standard input, standard output, and standard error.

This version corrects several bugs found in the original version printed in *Dr. Dobb's Journal,* December 1985 through March 1986 issues. It runs on any MS-DOS computer.

Add additional features to the Shell with

/util

A UNIX-like Utility Package for MS-DOS

This collection of utility programs for MS-DOS includes updates of the highly acclaimed Dr. Dobb's articles *Grep: A UNIX-Like Generalized Regular Expression Processor, Ls* from *Dr. Dobb's* C Chest column, and *Getargs* from DDJ's C Chest.

Source code is included and all programs (and most of the utility subroutines) are fully documented in a UNIX-style manual. You'll find executable versions of:

cat A file concatenation and viewing program

cp A file copy utility

date Prints the current time and date

du Prints amount of space available and used on a disk

echo Echoes its arguments to standard output

grep Searches for a pattern defined by a regular expression

Ls Gets a sorted directory

mkdir Creates a directory

mv Renames a file or directory. Moves files to another directory.

p Prints a file, one page at a time

pause Prints a message and waits for a response

printenv Prints all the environment variables

rm Deletes one or more files

rmdir Deletes one or more directories

sub Text substitution utility. Replaces all matches of a regular expression with another string.

Order The Shell and /util together for only $50! SAVE OVER 15%!

Item #160 The Shell	$29.95
Item #161 /util	$29.95
Item #162 Shell/util Package	$50.00

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or

CALL TOLL-FREE 1-800-528-6050 EXT. 4001

and refer to product item number, title, and disk format.

For customer service questions, CALL M&T PUBLISHING, INC. 415-366-3600 EXT. 209

M&T BOOKS

David E. Cortesi longtime Dr. Dobb's columnist and author of *Inside CP/M* brings you —

Dr. Dobb's Z80 Toolbook

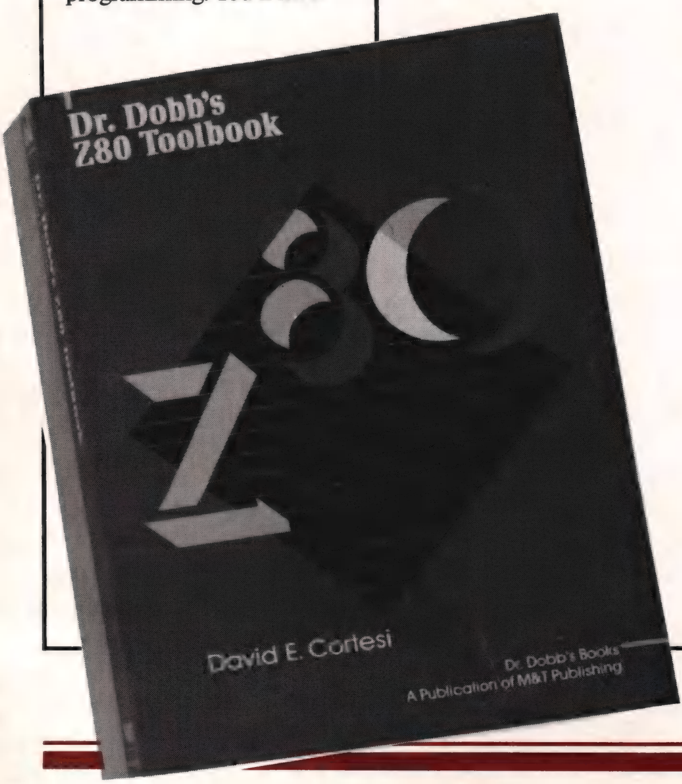
Dr. Dobb's Z80 Toolbook

Here's all you need to write your own Z80 assembly language programs for only $25!

Do you use CP/M? Do you feel as if the only part of the computer industry that has **not** abandoned you is your own Z80 computer? It keeps on working, but when you need programs for it, you have to write them yourself. When you do, you quickly find that while Pascal or BASIC is okay for some things, there is often no substitute for the speed, small size, and flexibility of an assembly language program.

Dr. Dobb's Z80 Toolbook puts the power of assembly language in the hands of anyone who's done a little programming. You'll find:

****A method of designing programs and coding them in assembly language.** Cortesi will take you on a walk through the initial specifications, designing an algorithm and writing the code. He demonstrates this method in the construction of several complete, useful programs.
****A complete, integrated toolkit of subroutines** for arithmetic, for string-handling, an for total control of the CP/M file system. They bring the ease and power of a compiler's runtime library to your assembly language work, without a compiler's size and sluggish code.
Best of all, every line of the toolkit's sorce code is there for you to read, and every module's operation is explained with the clarity and good humor for which Dave Cortesi's writing is known.



Order the Z80 Software on Disk! Save Yourself the Frustration of File Entry

All the software in **Dr. Dobb's Z80 Toolbook**—the programs plus the entire toolkit, both as sorce code and as object modules for both CP/M 2.2 and CP/M Plus—is yours on disk. (A Z80 microprocessor and a Digital Research International RMAC assembler or equivalent are required.) **Receive *Dr. Dobb's Toolbook for Z80,* along with the software on disk, together for only $40!**

Item #022 $25
Dr. Dobb's Toolbook for Z80
Item #022A $40
Dr. Dobb's Toolbook for Z80, together with software on disk. Please specify one of the following formats: 8″ SS/SD; Apple; Osborne; Kaypro.

To Order:

To order any of *Dr. Dobb's* products, return the order form at the end of this catalog, or
CALL TOLL-FREE
1-800-528-6050 EXT. 4001
and refer to product item number, title, and disk format.
For customer service questions,
CALL M&T
PUBLISHING, INC.
415-366-3600 EXT. 209

M&T BOOKS

Dr. Dobb's Catalog Order Form

ORDER NOW!

SOLD TO:

Name

Street (Please use street address, not P.O. box)

City State Zip

Day Phone ( )

For Faster Service on Credit Card Orders

Call Toll Free

1-800-528-6050 Ext. 4001

Please Refer to Item # When Ordering

Or, fill out this postage-paid, self mailing order form and return to: M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063.

For disk orders, please indicate format. Refer to ad for standard format availability for each product.

- ☐ MS/DOS
- ☐ CP/M ___Kaypro ___Apple ___Zenith Z-100 DS/DD ___Osborne ___8″ SS/SD
- ☐ Macintosh

Special formats available for additional $10 per product. Please inquire.

	Quantity	Item #	Description	Price Ea.	Total Price
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
				Sub Total	
				Sales Tax	
				Shipping	
				Total Order	

CA residents add applicable sales tax on merchandise total ___%___
(CA residents must add tax to all items EXCEPT *Dr. Dobb's Sourcebook,* Item #004) → Sales Tax

Shipping must be included with order. See rates below. → Shipping

☐ Check

Make checks payable to M&T Publishing, Inc.

☐ VISA

☐ MasterCard

☐ American Express

Name on Card

Account No.

Expiration Date

Signature

In U.S. For Bound Volumes, add $2.25 per book. Add $8.75 for special C Packages. For other books and disks, add $1.75 per item.

Outside U.S. For Bound Volumes, add $5.25 per book surface mail. Add $18 surface mail for Special C Packages. For other books and disks, add $3.25 per item surface mail. Foreign airmail rates available on request.

Prompt Delivery!
Dealer Inquiries Welcome

M&T BOOKS

3115S

Dr. Dobb's Catalog Order

Please Rush!

Please fold along fold-line and staple or tape closed.

No Postage
Necessary
If Mailed
In The
United States

BUSINESS REPLY MAIL

First Class Permit No. 790 Redwood City, CA

Postage Will Be Paid By Addressee

Dr. Dobb's Catalog
501 Galveston Dr.
Redwood City, CA 94063

Please fold along fold-line and staple or tape closed.

Write-Hand Man

"Almost a *Sidekick* for CP/M"

Ted Silveira—Computer Currents, Aug. 27, 1985

"WHM is ingenious and works as intended"

Jerry Pournelle, BYTE Magazine, Sept. 1985 (c) McGraw-Hill

Now available for CP/M 2.2, CP/M 3.0 and ZRDOS!

The convenience of *Sidekick* on your CP/M machine! Trigger **Write-Hand-Man** with a single keystroke and a window pops open to run desk accessories. Exit **Write-Hand-Man** and both the screen and program are restored. Use with any CP/M program and most any CP/M machine. Takes only 5K of memory.

FEATURES

- Notepad for quick notes
- Appointment calendar
- HEX calculator
- File and Directory viewer
- Quick access phonebook
- 14 digit decimal calculator

BONUS

Add applications written by you or others! No other *"Sidekick"* lets you add applications. Dump screens, setup printers, communicate with other computers, display the date and time. Let your imagination run wild!

$49.95 (California residents add tax), shipping included. COD add $2. Sorry, no credit cards or purchase orders. 30 day guarantee. Formats: 8 inch IBM, Northstar and most 5 inch (please specify).

Write-Hand-Man only works with CP/M 2.2, ZRDOS and CP/M 3.0 (please specify). Simple terminal configuration required. Not available for TurboDOS. Compatible with keyboard extenders, hard disks, and other accessories.

Poor Person Software
3721 Starr King Circle
Palo Alto, CA 94306
415-493-3735

Trademarks: **Write-Hand-Man** — **Poor Person Software**, CP/M—Digital Research, *Sidekick*—Borland International

Circle **no. 169** on reader service card.

Parallel Programming for "C"

TEAMWORK

A Concurrent Programming Toolkit

Teamwork is a "C" program library which allows you to write your programs as a set of cooperating concurrent tasks. Very useful for simulation, real-time applications, and experimentation with parallel programming.

FEATURES

- Supports a very large number of tasks (typically more than 50) limited only by available memory. Low overhead per task results in very fast context switching.
- Provides a full set of inter-task communication (ITC) facilities, including locks, semaphores, blocking queues, and UNIX*-style signals. Also has building blocks for constructing your own ITC facilities.
- Handles interrupts and integrates them into task scheduling. Supply your own interrupt handlers or block tasks on interrupts.
- Lets you trace task switches and inter-task communication.
- Comes with complete documentation including a user's manual and reference manual of commands.

Teamwork is available for the following systems:

Hardware	*Operating System*	*Price*
IBM PC, XT, AT	PC-DOS 2.0 or later	$ 65
IBM PC AT	XENIX*	$ 85
DEC VAX*	UNIX 4.2BSD	$195

PC-DOS version is compatible with DeSmet, Lattice, and Microsoft C compilers.

Please specify hardware and operating system when ordering. Send check or money order to:

Block Island Technologies
Innovative Computer Software
13563 NW Cornell Road, Suite 230, Portland, Oregon 97229-5892

*Trademarks: UNIX, AT&T Bell Laboratories, Inc.; XENIX, Microsoft, Inc.; VAX, Digital Equipment Corporation

Circle **no. 275** on reader service card.

OASYS Tool Kit Update

8088 8086 80186 **80286**

OUR INTEL SOFTWARE TOOL KIT KEEPS ON GROWING ...

WIZARD C-8086

- Supports 8086/88/87/186/286
- Runs in native and cross modes
- Complete K&R C plus V7 & III extensions
- Built-in LINT facility
- Addressing Models: small, compact, medium, large, huge
- Over 250+ functions in Run-time Library
- In-line assembly supported
- Intel & Microsoft compatibility at source and object levels
- Written in C; easily ported
- Highly optimized: 25%-40% tighter code
- Compiles over 1,000 lines/min.; 2+ times faster than Lattice
- Generates ROMable code
- Ideal for embedded real-time system or PC application development
- Supports DOS 2.x and 3.x, IBM/BIOS

OTHER OASYS 8086 TOOLS

- Symbolic C Source Level Debuggers
- Intel compatible Macro Cross Assembler/Linker/Locator
- Overlay PC Linker
- 8086/186 Simulator
- 80+ function Floating Point Library
- C Profilers
- QA tools
- C Interpreter with Dynamic Linking Loader
- C Run-time Analyzers
- C Documentor/Formatters
- Up/down Line Communication Tools
- Full screen, multi-window Editors/WPs
- Unix-like Tools for PC
- Software design tools

AVAILABILITY

Native: PC/XT/AT, all PC look-alikes, and DS-32 PC Co-Processor running MS/DOS, PC/DOS, and Xenix.

Cross: VAX VMS/Unix, Pyramid, Sun, Masscomp, Apollo, dozens more.

You name it . . .

OASYS provides a "One-Stop Shopping" service for more than 125 products running on, and/or targeting to, the most popular 32-, 16- and 8-bit micros and operating systems.

Oasys
A DIVISION OF XEL

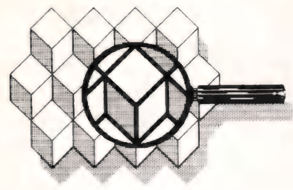
60 Aberdeen Avenue, Cambridge, MA 02138
(617) 491-4180

We Specialize In:
Cross/Native Compilers C, Pascal, Fortran, Cobol, Basic, APL, PL/1, Prolog, Lisp, ADA — Assemblers/Linkers — Symbolic Debuggers — Simulators — Interpreters — Translators Converters — Profilers — QA Tools — Design Tools — Comm. Tools OS Kernels — Editors — Spreadsheets — Data Bases — VAX & PC Attached Processors and more

We Support:
680xx, 80x86, 320xx, 68xx, 80xx, dozens more

Circle **no. 254** on reader service card.

OF INTEREST



The theme of this month's column is user-interface design.

The Research Group has announced SayWhat?!, a productivity tool designed to shorten the time required for screen design and at the same time increase the impact of screen displays for dBASE II and III, BASIC, and Turbo Pascal programs. It is an unprotected menuless system that allows users to create and edit displays on the screen without writing any programming commands or modules. The program requires an IBM PC/XT/AT or compatible with a minimum of 128K (384K required for dBASE III), DOS 2.0 or later, one disk drive, and a monochrome or color monitor. It retails for $39.95.

Wendin is offering three new products: Operating System Toolbox, PCVMS, and PCUnix. Operating System Toolbox allows users to modify and tailor operating system software. Full source code is available. All enhanced system services, paged memory management, concurrent input/output, and a multitasking scheduler can be used to construct powerful personal operating systems for the IBM PC/XT/AT. The price is $99. PCVMS is an operating system that provides multiple processes, networking software, and a command set, plus a set of VAX-like system services. The price with source code included is $49. PCUnix features utilities such as *at*, *grep*, *mail*, *nice*, and *who*. The price is $49.

For C

Gimpel Software has announced Amiga-Lint, a diagnostic facility for the C programming language that runs on the Commodore Amiga. Amiga-Lint analyzes C programs and reports on bugs, glitches, and inconsistencies. Among the errors reported on by Amiga-Lint are type inconsistencies across modules, parameter-argument mismatches, library usage irregularities, uninitialized variables, value-return inconsistencies, variables declared but not used, suspicious use of operators, and unreachable code. Amiga-Lint runs under Amiga's CL1 interface and is available for $98.

Retrieval Technology Corp. (RTC) has available the All-Hands-on C Video Workshop, a six-module, five-hour workshop designed to teach the full features of C.

Desktop A.I. has released a translator that allows users to move dBASE programs into C. The dBx Translator system includes a language translator for processing dBASE source code and a run-time library toolbox to replace the dBASE screen handler. The system is designed to work with any C database manager. In addition, applications can be moved to machines on which dBASE is not available, such as the AT&T 3B2 under Unix, Altos under Xenix, and Macintosh or Amiga systems. The package price ranges from $350 to $1,000, depending on configuration.

The C-Board is a stand-alone C language development system from **HiTech Equipment** that can be used on a single STD bus-compatible board. The C-Board requires a 5V 0.5A power source or unregulated 7–9V DC using an optional on-card regulator. The product features a CPU with parallel and serial I/O, timer/counters with PWN output, EPROMs and EEPROMs, a threaded interpreter, and interactive development. The OEM version of the card without memory devices or manuals costs $199. Volume discounts are available.

A PC- and VAX-hosted C cross-compiler supporting Intel's 8-bit MCS-51 microcontroller family is available from **Archimedes Software**. The C-based software development kit implements the proposed ANSI standard for C compilers and also comes with an assembler. The software is available for IBM PC/XT/AT or compatible systems equipped with at least 512K RAM and using MS DOS 2.0 or later. Initially, the software is also available for Digital Equipment's VAX/Unix minicomputer. The PC-based version is $851; the VAX/Unix version is $3,500. VAX/VMS and MicroVAX versions will also be available.

Alcyon has expanded its optimizing C compiler line to include cross-development versions for the IBM PC/XT/AT. The C68 compiler, priced at $795, produces optimized code for the M68000/010. The C68/020, priced at $995, produces optimized code for the M68020/68881. Minimum requirements are an IBM PC/XT/AT, 5-megabyte hard disk, 512K, and PC DOS/MS DOS 3.0 or later. In addition to the IBM PC, Alcyon's C compilers can be hosted on systems with Motorola VersaDOS, DEC VAX/VMX, and DEC VAX/Unix.

The C Trainer Interpreter from **Catalytix** provides users with an interactive interpreter for the full C language. The interpreter enables users to run C code without compiling it. Furthermore, the interpreter's design permits users to run program fragments without requiring that all functions and libraries be present at run time. The book *The C Trainer*, published by Prentice-Hall, accompanies the interpreter and combines a step-by-step tutorial with a separate reference section explaining many aspects of C. The tutorial's format is such that readers build upon existing programs that help them understand C in an interactive fashion. The C Trainer Interpreter runs on the IBM PC and compatibles, as well as the Macintosh. Versions are also available for Unix and for VAX/VMS computer systems.

CDEBUG (Version 2) from **Complete Software** is a portable C language debugger that shows all C objects according to their name and type and lets users set unlimited break and trace points by using regular expressions. Requiring minimal overhead, the interactive program consists of a preprocessor to insert symbol tables into source code and a run-time library to interpret and integrate with a user environment. CDEBUG is available for MS DOS, Lattice, Computer Innovations, Wizard, VAX/VMS, Unix, and Xenix operating systems on disk or tape media and will be ported. The new version is

priced from $350 for PCs and $750 for the Unix porting service.

Fortrix-C from **Rapitech Systems** is designed to recognize and convert most FORTRAN VMS extensions and translate a typical 50,000-line program into C code on VMS. All comment lines remain in place so internal documentation is retained.

For the IBM PC

Midwest Micro-Tek has released Circuit Design Mate, a software product for the IBM PC and compatibles that includes schematic capture, automatic parts list generation, TTL component library and editor, and schematic printing on any Epson-compatible dot-matrix printer. The system requires an IBM PC/XT/AT or compatible with 256K, a 640 × 200 IBM-compatible graphics card, two double-sided disk drives or one disk drive and hard disk, and PC DOS/MS DOS 2.0 or later. A single copy with TTL library is $295.

Macmillan Software has unveiled a menu-driven software package for PCs. Called Asystant Ready-to-Run Scientific Software, the package is a high-level programming language with acquisition, analysis, and graphics capabilities. Asystant runs on IBM PCs and compatibles, including the Hewlett-Packard Vectra, and uses the 8087 coprocessor. A second version of the product, called Asystant +, adds data acquisition and includes built-in interactive data manipulation, analysis, and high-resolution color graphics. Asystant is priced at $495, and Asystant + costs $895.

Dasoft Design Systems' PC2 features an auto-router, expandable symbol library, enhanced footprint editor with six pad shapes and user-definable pad layouts, and a component "data book" library of 600 commonly used parts. The PC2 can be used on the IBM XT/AT and compatibles with 512K, a monochrome graphics card, hard-disk storage, and a mouse. The software also runs on the AT&T 6300 and the NEC 9801. It drives most plotters including those from Hewlett-Packard, Houston Instruments, Western Graphtek, Ioline, and compatibles. The software is available for $3,495.

The PC-NTDS Adapter Card, a full 32-bit parallel NTDS (MIL-STD 1397) interface board, is available from **Sabtech Industries**. The adapter installed in an IBM PC/AT utilizes the full 16-bit data path of the AT and can output data at speeds of up to one 32-bit word every 1.2 microseconds. This represents a transfer rate of up to 834K 32-bit words per second. Software included with the NTDS Adapter Card has its own interface control language, including high-level commands such as *loop*, *repeat*, and *compare*.

Utilities

Sophco has released Sybil, a collection of utilities that can unerase files, edit sectors, modify file and directory attributes, and find files. Sybil comes with a print spooler, a RAMdisk, a general-regulation expression parser, and a graphics editor called ASCIIGEN (for RGB and IBM monochrome). It runs on the IBM PC/XT/AT and compatibles and on both 8088 and 80286 computers. Sybil is priced at $49.95.

Avocet Systems has released AVSIM09, a software simulator/debugger for the 6809 microprocessor. Running on any IBM PC

(continued from page 123)

look-alike, AVSIM09 interpretively executes 6809 object code under control of a full-screen symbolic debugger. Devices supported include the 6821 PIA (Parallel Interface Adaptor), 6840 PTM (Programmable Timer Module), and 6850 ACIA (Asynchronous Communications Interface Adapter). AVSIM09 is priced at $299.

Modula-2 has been added to the list of languages recognized by Source Print, **Aldebaran Laboratories'** multifunction, source-code formatting utility. For Modula-2, Source Print can draw lines to indicate module and procedure nestings as well as nestings of *if*, *while*, *for*, *case*, *loop*, and *repeat* structures. Automatic indentation can also be performed. Additional features include facilities for indexes, tables of contents, pagination, line numbering, keyword emphasis, extraction of routines, and extensive printer control. Source Print is for IBM PC/XT/AT and compatibles. The source formatting utility with 88-page manual is $97.

Written in RM COBOL, Myte Myke Software from **M & D Systems** supports both single-user and multiuser microcomputers as well as the more advanced local-area networks including Novell and is designed to take advantage of the expanding computing power of various microcomputing systems. The software can be used with Unix Version V on Sperry 5000, AT&T 3B Series, and NCR Tower Installations.

ZLKO from **Elliam Associates** is used to link relocatable object files created by ZMAC or other assemblers that use the Microsoft REL format. ZLKO can be used with compilers for FORTRAN, BASIC, COBOL, or C. ZLKO links the program on disk rather than in memory and thus links a program that will fill the entire available memory space. It produces a symbol table for debugging and can be used to segment a program that will not fit into memory into a tree structure of overlays. The price for the disk and instruction manual is $49.95.

CompuMagic has released a package of 20 superutility programs for anyone using CP/M 2.2. There are three sets of programs. The File Management programs CMCopy, Erase, Rename, Compare, Sort, DoubleSpace, and WordCount allow wildcards, multiple commands on a line, and ask-me and test options, in addition to user-area switching. The Director programs MDIR and MDIRS take multiple file specifications. DIRBAK provides a list of all backup files, and DIRSPACE reveals how many directory entries are left. UDIR lists all the files in user areas, and DiskDIF tells what files are on one disk but not another.

In the Special Utilites programs, CMAuto allows the creation of programs to run other programs; TYPIT turns a computer into an electronic typewriter; and A.COM corrects the common *A;program* typo and turns it into *A:program*. Screen puts everything on the screen into a file, and R/O and R/W are replacements for STAT's command to convert files to read/only or read/write. Miniera is a 1K program that allows erasing. The complete package is $45.

Computer-Guru has introduced Salt & Pepper, a software program that contains 29 subroutine modules in MS DOS-compatible BASIC. The modules are saved in ASCII format and can be lifted from disk and merged into a user's program. The programmer can then use single-line commands to accomplish tasks such as creating professional menus and input screens, processing dates, changing strings to uppercase or lowercase, creating "walking" strings, trapping errors, and issuing Caps-Lock on/off. The package is available for $59.95.

UX Software has announced the UX-BASIC Native Code Compiler to complement the UX-BASIC Interpreter (Version 2.1). UX-BASIC features structured code; modular programming; sequential, direct, and ISAM files; and editing and debugging tools. It is functionally compatible with IBM's IX BASIC. Prices vary according to the class of target computer.

Software Products & Services (SPS) has expanded its EPOS engineering and software development environment to include tools for the automatic generation of Pascal code. The EPOS software system supports project design and development from the formulation of requirements to a complete design and system maintenance throughout the entire life cycle of a project. It also contains integrated management tools for project control. The system is language-independent and supports seven design methodologies.

For Apple

SuperMac Technology has introduced three Macintosh add-on boards. Enhancements include Meg, a 1-megabyte memory expansion board designed to coexist with internal hard disks; SuperDrive 20, a high-performance, 20-megabyte, 3½-inch internal hard disk designed to improve the Macintosh's file access speed up to ten times; and Enhance, a clip-on board that gives the original Macintosh 2 megabytes of contiguous RAM and a Macintosh Plus-compatible Small Computer System Interface (SCSI) port. Meg is available for $849 for upgrading from 128K and for $699 for upgrading from a 512K Macintosh. SuperDrive costs $1,299. Enhance can be expanded to 4 megabytes or more; its price has not yet been announced.

ProFiler 2.1, a data manager/report generator for Apple II series computers, now has its utility program integrated into the overall program. The utility feature allows transfer of data between ProFiler and AppleWorks. ProFiler 2.1 can design, organize, file, search, sort, merge, calculate, and print reports and run on either floppy or hard disks. The menu-driven program can store up to 1,500 records on a floppy disk or up to 65,000 records on a hard disk. No additional data entry is required to transfer data from one medium to another. The entire program, including the utility program, sells for $99.95 and is available from **PM Software**.

Transwarp, an accelerator card that speeds up both the main and auxiliary memory of an Apple IIe computer, is available from **Applied Engineering**. The accelerator works with all Apple II+ and IIe software, including AppleWorks, SuperCalc 3a, and VisiCalc, and is compatible with all standard peripheral cards. The $279 board plugs into any available slot in Apple II, II+, and IIe computers.

Odesta has unveiled

four Helix products for the Macintosh Plus. Helix is a database information-management and decision-support system that allows individuals to build applications tailored to their specific needs. Double Helix allows users to create custom menus and install security to protect the structure of their applications. Value-added resellers and application publishers can build stand-alone vertical packages with Run-Time Helix. MultiUser Helix allows networked users to work simultaneously with a customized Helix application. Helix and Double Helix for the Macintosh Plus and 512K Macintosh are available for $395 and $495, respectively. Run-Time Helix for both Macintosh products can be licensed for a fee of $500 for ten applications. The price has not yet been determined for MultiUser Helix.

Communications

Practical Peripherals' Modem 1200 is Hayes-compatible and supports pulse or touch-tone dialing in full- or half-duplex operation. It self-adjusts to varying transmission speeds from 300 to 1,200 bits per second. The product conforms to all Bell 212A and 103 standards and complies with the requirements of FCC Part 68 for direct connection to public phone lines. The 4 × 5-inch card is installed in either a short or long slot within IBM PC/XT/AT or compatible computers.

The Dual Serial Port Manager (DSPM) is a hardware/software interface program. Available from **Akron Software**, DSPM has a full-interrupt drive, automatically buffers all received and transmitted data, and allows users to specify the size of each buffer independently. It also allows an application program to use both COM1 and COM2 simultaneously and supports three data-transfer protocols. The DSPM package provides interfaces to programs written in Pascal, C, compiled or interpreted BASIC, FORTRAN, and assembly language. It costs $99.

Quasitronics has announced the Pipe Six, an intelligent data switch that has six RS-232 I/O ports that provide communications between dissimilar systems and/or peripherals operating with different baud rates, word structures, and flow control techniques. The system features 54K of allocated memory for speed conversion, printer buffering, message storage, and private mail. It can handle concurrent communciations between all of its fully interactive ports.

Unlimited Processing has announced Team-Up, a Total Environment for Application Management. Team-Up adds application management to data management functions by managing up to 32,000 applications containing up to 700,000 files. It tracks file existence, file names, server/directory file locations, and user-access authorizations. Prices range from

OF INTEREST
(continued from page 125)

$795 to $4,495, depending on configuration.

Miscellaneous

USecure, a system security and administration product for Unix-based computers, is available from **Unitech Software**. USecure is menu-driven and provides automated system access control and audit trails of system changes, file modifications, access permissions, deleted files, and start-up and shutdown activity. Pricing varies from $300 on a PC to $2,500 on a mainframe.

Logicraft's GraVAX brings high-scale resolution graphics and PC compatibility to the VAX environment while maintaining the VT-100 DEC terminal capabilities. GrafVAX is a bus extension of the company's Cardware product line and consists of two components. The Q-bus version, which plugs directly in the DEC Q-bus, is priced at $2,990 per user, and the Unibus version, which can be located up to 4,000 feet from the CPU, is priced at $5,980 per unit.

The MII-1 Bubbl-Board, a bubble-memory system from **Bubbl-tec** allows Multibus II machines to make use of solid-state mass storage in applications for which electromechanical media such as disk and tape are unsuitable. The MII-1 system provides 512K of nonvolatile mass storage on a singlewide Multibus II module and incorporates an intelligent controller that handles device formatting and control, interfaces the bubble-memory system to the Multibus II bus structure, and provides for both soft- and hard-error detection and correction. The system is also expandable to 32 megabytes. The 512K version is priced at $2,899 in quantities of ten. Versions with smaller amounts of onboard storage capacity are also available.

Reference Map

Akron Software, 53 Hillside Ave., Toronto, Ont., Canada M8V 1S7; (416) 251-1866. Reader Service Number 16.

Alcyon Corp., 5010 Shoreham Pl., San Diego, CA 92122; (619) 587-1155. Reader Service Number 17.

Aldebaran Laboratories Inc., 3738 Mt. Diablo Blvd., #312, Lafayette, CA 94549; (415) 283-7084. Reader Service Number 18.

Applied Engineering, P.O. Box 798, Carrollton, TX 75006; (214) 241-6060. Reader Service Number 19.

Archimedes Software Inc., 1728 Union St., San Francisco, CA 94123; (415) 771-3303. Reader Service Number 20.

Avocet Systems Inc., 120 Union St., P.O. Box 490, Rockport, ME 04856; (800) 448-8500. Reader Service Number 21.

Bubbl-tec, 6800 Sierra Ct., Dublin, CA 94568; (415) 829-8700. Reader Service Number 22.

Catalytix Corp., 55 Wheeler St., Cambridge, MA 02138; (617) 497-2160. Reader Service Number 23.

Complete Software Inc., 60 Aberdeen Ave., Cambridge, MA 02138; (617) 492-5305. Reader Service Number 24.

CompuMagic Inc., P.O. Box 437, Severn, MD 21144; (301) 969-8068. Reader Service Number 25.

Computer-Guru, 40 Wagner Ave., Piscataway, NJ 08854; (201) 356-6477. Reader Service Number 26.

Dasoft Design Systems Inc., 2550 Ninth St., Ste. 113, Berkeley, CA 94710; (415) 486-0822. Reader Service Number 27.

Desktop A.I., 1720 Post Rd., East, #3, Westport, CT 06880; (203) 255-3400. Reader Service Number 28.

Elliam Associates, 24000 Bessemer St., Woodland Hills, CA 91367; (818) 348-4278. Reader Service Number 29.

Gimpel Software, 3207 Hogarth Ln., Collegeville, PA 19426; (215) 584-4261. Reader Service Number 30.

HiTech Equipment Corp., 9560 Black Mountain Rd., San Diego, CA 92126; (619) 566-1892. Reader Service Number 31.

Logicraft Inc., 410 Amherst St., Nashua, NH 03063; (603) 880-0300. Reader Service Number 32.

Macmillan Software Co., 630 Third Ave., New York, NY 10022; (800) 348-0033, in NY (212) 702-3241. Reader Service Number 33.

M & D Systems Inc., 3885 N. Buffalo Rd., P.O. Box 108, Orchard Park, NY 14127; (716) 662-6611. Reader Service Number 34.

Midwest Micro-Tek Inc., P.O. Box 29376, Brooklyn Center, MN 55429; (612) 560-6530. Reader Service Number 35.

Odesta Corp., 4966 El Camino Real, Los Altos, CA 94022; (415) 962-8661. Reader Service Number 36.

PM Software, P.O. Box 1788, Huntington Beach, CA 92647; (714) 963-2221. Reader Service Number 37.

Practical Peripherals Inc., 31245 La Baya Dr., Westlake Village, CA 91362; (800) 641-0814. Reader Service Number 38.

Quasitronics Inc., 211 Vandale Dr., Houston, PA 15342; (800) 245-4192. Reader Service Number 39.

Rapitech Systems Inc., 75 Montebello Rd., Suffern, NY 10901; (800) FORTRIX, in NY (914) 368-3000. Reader Service Number 40.

Research Group (The), 813 W. Grant Pl., San Mateo, CA 94402; (415) 571-5019. Reader Service Number 41.

Retrieval Technology Corp. (RTC), 3 Courthouse Ln., Chelmsford, MA 01824; (617) 458-1130. Reader Service Number 42.

Sabtech Industries, 4091 E. La Palma Ave., Unit P, Anaheim, CA 92807; (714) 630-9335. Reader Service Number 43.

Software Products & Services (SPS), 14 E. 38th St., 14th Floor, New York, NY 10016; (212) 686-3790. Reader Service Number 44.

Sophco Inc., P.O. Box 7430, Boulder, CO 80306-7430; (800) 922-3001. Reader Service Number 45.

SuperMac Technology, 1901 Old Middlefield Wy., Mountain View, CA 94040; (415) 964-8884. Reader Service Number 46.

Unitech Software, 8330 Old Courthouse Rd., Ste. 800, Vienna, VA 22180; (703) 734-9844. Reader Service Number 47.

Unlimited Processing Inc., 8382 Baymeadows Rd., Ste. 8, Jacksonville, FL 32216; (800) 874-8555. Reader Service Number 48.

UX Software Inc., 10 St. Mary St., Toronto, Ont., Canada M4Y 1P9; (416) 964-6909. Reader Service Number 49.

Wendin, P. O. Box 266, Cheney, WA 99004; (509) 235-8088. Reader Service Number 50.

—Wendelin Colby

DDJ

ADVERTISER INDEX

*This advertiser prefers to be contacted directly: see ad for phone number.

Advertising Sales Offices

East Coast
Walter Andrzejewski (617) 868-1524

Midwest
Michele Beaty (317) 875-8093

Northern California/Northwest
Lisa Boudreau (415) 366-3600

Southern California/AZ/NM/TX
Michael Wiener (415) 366-3600

Advertising Director
Shawn Horst (415) 366-3600